Towards Integrated Failure Recovery for Web Service Composition

Paul Diac[1], Emanuel Onica[1]
[1]*Faculty of Computer Science, Alexandru Ioan Cuza University of Iaşi, Romania*

Keywords: Web Service Composition, Fault Tolerant Systems, Failover Mechanism, Reconstruction Algorithms.

Abstract: Web Service Composition (WSC) aims at facilitating the use of multiple Web Services for solving a given task. Automatic WSC is particularly focused on automating the composition process following specifications of involved actors. There are many initiatives developed in this direction, but most of them stop at describing the process of building the composition, while some also study the coordination of the execution. However, research of specific solutions in cases of failure of services is typically not integrated with composition algorithms and mostly relies on costly additional measures. Fault tolerance is important for developers relying on the compositions, since their applications are rarely for transient usage, requiring high availability over time. Naively re-computing the compositions from scratch can be a waste of resources for the composition engine. In this paper, we propose addressing web service faults by preparing backup compositions as part of a fallback mechanism, which efficiently integrates with the base composition algorithm.

1 INTRODUCTION

In the popular Service Oriented Architecture (SOA), web services are the building blocks on top of which most distributed applications are built. Although already a well-established paradigm, SOA importance is still on the rise with the exploration of new aspects, like IoT, micro-services, and semantic services. A web service is an atomic component, which provides a specific functionality declared in its definition. This definition can be strictly syntactic, or enhanced with semantic descriptors or other types of meta-data information, for example, Quality of Service metrics. In general, one single web service has a relatively simple functionality, bound to a specific application domain, providing some output value(s) for some input parameters (e.g., providing an output result following a particular input query on a database).

Web Service Composition (WSC) permits aggregating the use of multiple services into more complex applications. Whenever the desired output information cannot be obtained by means of a single service, a composition algorithm can be executed, connecting multiple web services by using intermediate outputs as inputs for other services, until the result is found. Multiple techniques have been proposed for creating such compositions focusing typically on performance aspects. Not many of these take into account fault tolerance. This is generally regarded as an orthogonal problem that is solved by other means independent of the composition technique. Web service failure is a real issue with multiple causes, some of which, like failures of infrastructure or failures in communication (MoreThanCoding.com, 2012), completely disable the use of a service. Most failover mechanisms either consider only the case of execution malfunctioning where the service can be restarted or rolled back to a previous state, or focus strictly on the integration of redundancy measures in case of complete failures. There is, however, less focus on how to obtain such needed redundancy, which can serve for a re-composition. The approach we propose takes into account the possibility of complete service failures, integrating the needed redundancy for efficient recovery from the initial composition stage.

In Section 2 we present the composition model. We describe the solution for failure recovery in Section 3. In Section 4 we empirically evaluate our proposed algorithms on a synthetic, automatically generated repository. In Section 5 we discuss related work on providing fault tolerance for Web Service Composition. Finally, Section 6 concludes our contribution and proposes several future work possibilities.

2 COMPOSITION MODEL

Applications using Web Service Composition typically consider three types of actors: the service providers, the users requesting the compositions for some functionality, and the composition generator. Very often the composition generation is represented

by a static method applied only once, which generates the composition and returns it to the user. Nevertheless, we expect that most often the user would need that functionality for a continuous period of time, over which the composition should remain available. Our design setting considers a *composition engine* as an application that can dynamically respond to users requests for new compositions, or handling services downtime, which is the focus of our paper.

An early effective composition model was provided in the first Web Services Challenge (Blake et al., 2005). The challenge required solving specific user queries by implementing an automatic computation of a web services composition. The repository of services consisted of WSDL (Christensen et al., 2001) files with service definitions. Solutions had to output valid compositions and were evaluated by their runtimes over a batch of tests. Another important evaluation metric is the composition length. If multiple valid compositions exist, the shorter is preferred. This length optimization has both performance and fault-tolerance implications (i.e., fewer services in a composition makes it less prone to failure). Generally, finding the shortest composition is an `NP-Hard` problem, as proven in (Ţucăr and Diac, 2018). (Diac, 2017) proved to be significantly faster than other proposed algorithms in the Web Services Challenge 2005 benchmark. The composition model in this solution serves as the basis for our current work.

We formally summarize in the following the elements of the model and present its basic mode of operation.

Parameters. Let $\mathbb{P}$ be the set of all parameters, identified by names, that appear in any service definition or in the user request. $\mathbb{P}$ is not restricted in any way, and it can include any string.

Web Services. A service is identified by a name and contains two sets of parameters, input and output: $ws_i = \langle name, In, Out \rangle$, where $In, Out \subseteq \mathbb{P}$.

Service Repository. The service repository, written as $\mathbb{R}$, is the set of all services, $\mathbb{R} = \bigcup_{\forall i} ws_i$.

Composition Query. A *user request* for a composition or a *composition query* can be thought of as a service that does not exist yet, but the user has interest in. It is defined over a set of input parameters that the user initially knows and a set of requested (output) parameters that the user needs. All active queries $q_x = \langle name, In, Out \rangle$; $In, Out \subseteq \mathbb{P}$ make the set $\mathbb{Q}$.

Valid Composition. For query q_x, a valid service composition, is a sequence of services $\langle ws_1, ws_2, ...ws_k \rangle$, where: each service input set must be included in the set of all previously called services outputs, or in the user initially known parameters, and the final user query output must be covered by services outputs.

$$ws_i.In \subseteq \left(q_x.In \cup \left(\bigcup_{j=1}^{i-1} ws_j.Out \right) \right), \ \forall \ i = \overline{1..k}$$
$$and \quad q_x.Out \subseteq \left(\bigcup_{i=1}^{k} ws_i.Out \right)$$

Mode of Operation. The composition construction gradually builds a sequence of services structured as a directed acyclic graph. Each service depends on some of the previously selected services, meaning its input parameters are bound to the output parameters of previous services. These parameters are matched simply by name: an output of some service can be used as input of another service if their names coincide. Simple typing can slightly extend this model. Hierarchical types, i.e., concepts arranged in a taxonomy - defined the first step towards semantic composition in (Weise et al., 2008). As the focus of this paper is on the failover mechanism we consider the model limited to name equality that can be extended.

When selecting a service in the composition, there can be multiple options that generate one particular needed parameter, so there can be a choice of what service to use. A service that is fit but is not chosen can serve as a backup starting point, for the case of an eventual failure of the initially selected service. The composition algorithm can prepare for this backup ahead of time, improving the overall performance perceived by the user. In Section 3 we present in detail our algorithms providing backup, enhancing the model for use-cases where faults appear at the execution phase of the composition.

3 FAILURE RECOVERY

The algorithm we propose integrates with a failover mechanism, and it is based on a traversal over the set of *reachable* parameters, where services are used to find new parameters for building the composition. A heuristic score is assigned to services that is used when multiple services can be chosen.

Our approach also handles dynamic removal of a service at the provider's request, which has similar implications with complete service failures at execution time.

3.1 Overview

The composition algorithm is built upon an adjusted breadth-first search traversal over the set of parameters. This starts from the user known parameters, aiming to obtain the user required parameters. A loop tries to select the next services to call if any is available. Learned output parameters are marked as such in a set, and for each service, all unknown input parameters are kept in a map. When one such map becomes empty, the corresponding service becomes available to call. The loop stops when all user required parameters are learned. If multiple services become available at the same time, the one with the best score is selected. This score approximates the usefulness of the service and is the length of the shortest possible distance from the service to any of the final user required parameters (distance is calculated in the number of services). When selecting a service, there is no guarantee that the service will be useful in the final composition. Therefore, as a final step, we also execute a composition shortening algorithm. This shortening algorithm navigates the services in the reverse order of the composition and deletes all services that do not output used parameters.

The design of the solution is to return the composition as soon as constructed to the user in order to provide a fast response time. To provide the failover mechanism, the algorithm pre-computes alternative compositions for each user query. These alternative compositions are searched in a background thread. Each backup is stored internally at the time it completes. If a service fails, the backup is provided to the user as the new active composition, and immediately takes an active role instead of the disrupted composition. Then, similarly, in the background, a new backup to the new active composition is re-computed, and the old one is deleted. It is expected that these backup compositions are computed initially before any service failure. However, in the other, unlikely and worst case situation, it is easy to detect that the composition is broken without having a backup ready for replacement. If this happens, the composition is re-built from scratch as for a new query, and the user will just notice a longer response time.

To compute the alternative backup compositions, the algorithm prepares for cases in which each (but only one) service goes down. If more services break at the same time, the recovery can take longer, as the solution is to process failed services sequentially.

3.2 Failover Mechanism Details

The response composition for a query is represented as a simple ordered sequence of services. The order defines chronological execution of services and is determined according to how parameters matched between services. If one of the services fails, it might be replaceable by other services in the repository. To find a solution equivalent to the failed service, we search for a new backup composition. This search is similar to solving a new user query, and we use again the same composition search algorithm. However, we need to prepare new appropriate parameter sets. This is depicted in Figure 1, where ws_p breaks. All the services that are situated before the failed service provide a set of $\{known\}$ parameters, which includes the user's initially known parameters. These will be the input parameters of the new query. We prepare the output parameters targeting to keep in the new composition the still valid services from the old suffix, after the failed service. All new required (output) parameters are collected in the $\{required\} \setminus \{known\} \setminus \{gen\}$ set, where $\setminus$ is the set difference operator. The $\{required\}$ set contains all input parameters of any service on the right of ws_p and any initially user required parameter. It is obvious that we can remove the $\{known\}$ parameters out of this new query output set. Parameters in $\{gen\}$ are inputs of subsequent services that are also "generated" as outputs of other (previous) services on the right. If we consider the old suffix after ws_p still valid and try to keep it, $\{gen\}$ parameters are not actually necessary. We can also remove them from the new required set. The reason for building the sets in this manner is that the more parameters are added to the known and the less are required, the shorter the composition will be.

If no composition is found for the new ($\{known\} \rightarrow \{required\} \setminus \{known\} \setminus \{gen\}$) query, it does not necessarily mean that no solution exists. There might be the case that services succeeding ws_p use some parameters that are no longer accessible. In such a situation, all the composition's suffix, i.e., services on the right of ws_p must be reconstructed. This, again, can be done by a new composition query: ($\{known\} \rightarrow \{user\ requested\} \setminus \{known\}$), displayed as the *suffix query* in Figure 1.

If this does not succeed either, the user query is no longer solvable, because only keeping the services on the left of ws_p can be at most useless, but not wrong.

Regardless of how the composition was reconstructed, the algorithm does not maintain the backups for the "partial" compositions themselves (i.e., the ones replacing just ws_p or the suffix of the initial composition). Backup solutions are built for

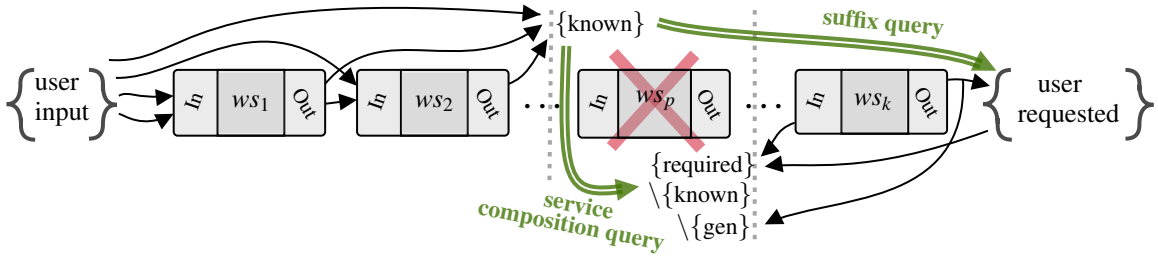

Figure 1: Service $ws_{p \leq k}$, part of composition $\langle ws_1, ws_2, ..., ws_k \rangle$ fails. ws_p could be replaced by a new composition that uses all parameters learned from outputs of any previous service (on the left), and that generates all parameters that are at input on the right. If it is not found, we can drop the services on the right of ws_p and try an alternative composition replacing all the "suffix" that can potentially be affected by the failed service.

the whole new composition. The reduction algorithm is run on the new composition, eliminating useless or duplicate services that can appear.

Another action taken when a service fails is to rebuild all backup compositions that use it. These are backups for solutions of other user queries that are still working, i.e., that do not contain the failed service in the main composition but only in their backups.

3.3 Data Structures Used in Algorithms

We summarize in the following the data structures used in our proposed failover algorithms. We refer here only to the main global structures that keep high-level information like the services, the user queries and the solutions with their backups:
→ Set⟨Service⟩ **repository**: set $\mathbb{R}$ of all services;
→ Map⟨Parameter, Set⟨Service⟩⟩ **inputFor**: all services that have a specific parameter as input;
→ Set⟨Query⟩ **requests**: set $\mathbb{Q}$ of user queries;
→ **Query** objects contain: **.In** and **.Out** : user known and requested parameters;
→ Map⟨Query, Solution⟩ **compositions**: solutions;
→ **Solution** objects contain: Query **.query**, the resolved query; Array⟨Service⟩ **.main**, services in solution; Map⟨Service, Solution⟩ **.backup**, backup solution for each used service and Solution **.parent**, the composition it is backup to if *.main* is a backup;
→ Map⟨Service, Set⟨Solution⟩⟩ **usages**: main or backup compositions using some service.

3.4 Failover Algorithms

Our first algorithm functions presented in Algorithm 1 create a composition for a user query. This integrates with computing the needed backups for failure situations described in Algorithm 2. The functions of the algorithms use a series of temporary structures instantiated per query: **known**, **required**, **score**, and **ready**.

Algorithm 1 Find composition and learning.

```
1:  Set⟨Parameter⟩ known; // known parameters
2:  Map⟨Service, Set⟨Parameter⟩⟩ required; // input
    parameters of each service that are not yet known
3:  Map⟨Service, Int⟩ score; // heuristic service score
4:  Set⟨Service⟩ ready; // services with empty re-
    quired.get(), so callable; ordered by scores.
5:
6:  function FINDCOMPOSITON(query)
7:    solution ← new Solution();
8:    solution.query ← query; known, ready ← ∅;
9:    score ← COMPUTESERVICESCORES(query);
10:   for all service ∈ repository do
11:     required.put(service, service.In);
12:   LEARNPARAMETERS(query.In);
13:   while (¬ ready.empty() ∧ query.Out ⊈ known) do
14:     nextService ← ready.first(); // best scoring
15:     solution.main.add(nextService);
16:     LEARNPARAMETERS(nextService.Out);
17:   if (query.Out ⊈ known) then
18:     return NULL; // query is unsolvable
19:   else
20:     REMOVEUSELESSSERVICES(solution, query);
21:   for all (service ∈ solution.main) do
22:     usages.get(service).add(solution);
23:   return solution;
24:
25: function LEARNPARAMS(Set⟨Parameter⟩ pars)
26:   for all (service ∈ (repository \ ready)) do
27:     required.get(service).removeAll(pars);
28:     if (required.get(service).isEmpty()) then
29:       ready.add(service); // just became callable
30:   known ← known ∪ pars;
```

Their exact role is explained at the beginning of the listings. The FINDCOMPOSITION() function is used to find the main, first solution for any composition query. This makes use of LEARNPARAMS()

helper function, which has the role to maintain the known parameters set up-to-date during the algorithm's execution. The function FINDBACKUP() is responsible with computing the backup for one component service in a composition. The functions are wrapped in BACKUPCOMPOSITION(), which is essentially the function that the user should call for executing a composition with failover support.

Algorithm 2 Finding backup alternatives.

```
1:  function BACKUPCOMPOSITION(query)
2:    solution ← FINDCOMPOSITION(query);
3:    for all (service ∈ solution.main) do
4:      bkp ← FINDBACKUP(solution, service);
5:      solution.backup.put(service, bkp);
6:
7:  function FINDBACKUP(sol, service)
8:    p ← sol.main.indexOf(service);
9:    known_bkp ← sol.query.In;
10:   for (i ← 0 to p - 1) do
11:     known_bkp ← known_bkp ∪ sol.main.get(i).Out;
12:   req_bkp ← sol.query.Out;
13:   for (i ← sol.main.length down-to p + 1) do
14:     req_bkp ← req_bkp \ sol.main.get(i).Out;
15:     // remove {gen} from Figure 1
16:     req_bkp ← req_bkp ∪ sol.main.get(i).In;
17:   s ← FINDCOMPOSITION((known_bkp, req_bkp));
18:   if (s == NULL) then
19:     query ← (known_bkp,sol.query.Out\known_bkp);
20:     s ← FINDCOMPOSITION(query);
21:     // second type or "suffix-query" backup
22:   if (s ≠ NULL) then   s.parent ← sol;
23:   return s;
```

For simplicity, we omitted the listing of some helper functions. COMPUTESERVICESCORES() computes scores indicating how good services are to be part of the composition. The function performs a reversed order traversal of all services, starting from services that output any of the user's required parameters. These get the best score which is *1*. Following, services that output any input of services with score *1* get a score of *2*, and so on. Later on line 14 of Algorithm 1, the service with the best, lowest value score, is chosen out of all services currently available and added to the composition. The REMOVEUSELESSSERVICES() is another helper function. This is referenced in Section 3.1 as the shortening algorithm and is used for cleaning the composition of any services that are not mandatory.

Several optimizations can be applied to our algorithm construction. It is easy to observe that the calls to FINDBACKUP() for finding backups to each of the services in the composition can be executed in parallel. Otherwise, an improvement can also be obtained by building $\{known_{bkp}\}$ and $\{required_{bkp}\}$ sets only once instead of for each FINDBACKUP() call.

In Algorithm 3 we describe the delete service method. This is called when a service provider specifically deletes a service or when a service fails. Both cases are treated similarly triggering replacement with a backup and a new backup re-computation.

Algorithm 3 Remove service method.

```
1:  function DELETESERVICE(ws)
2:    repository.remove(ws);
3:    for all (param ∈ ws.In) do
4:      inputFor.get(param).remove(ws);
5:    for all (sol ∈ usages.get(ws)) do
6:      if (sol.parent == NULL) then // main solution
7:        bkpSol ← sol.backup.get(ws);
8:        sol.backup.clear(); // also update usages
9:        compositions.put(sol.query, bkpSol); // swap
10:       if (bkpSol ≠ NULL) then
11:         for all serv ∈ bkpSol.main do
12:           newBkp ← FINDBACKUP(bkpSol, serv);
13:           bkpSol.backup.put(serv, newBkp);
14:       // else sol.query becomes unsolvable
15:     else // sol was a backup, try to find another
16:       newBkp ← FINDBACKUP(sol.parent, ws);
17:       sol.parent.backup.put(ws, newBkp);
18:   usages.get(ws).clear();
```

4 EVALUATION

We evaluated our algorithms on synthetic generated data. We implemented a specific tests generator.

First, several queries are generated, each from two disjoint sets of random parameters: input and output, i.e., initially known and requested parameters.

For each of these queries, we build the resolving composition as a sequence of services modified to have each input chosen randomly from the outputs of previous services or from the query's input parameters. To create backup possibilities, we build an alternative solution for a randomly chosen service of the main solution. The alternative solution can either replace just the chosen service - the *first* type of backup or also all it's successive services - the *second* type of backup. In Figure 2, gray edges show how parameters pass through a composition

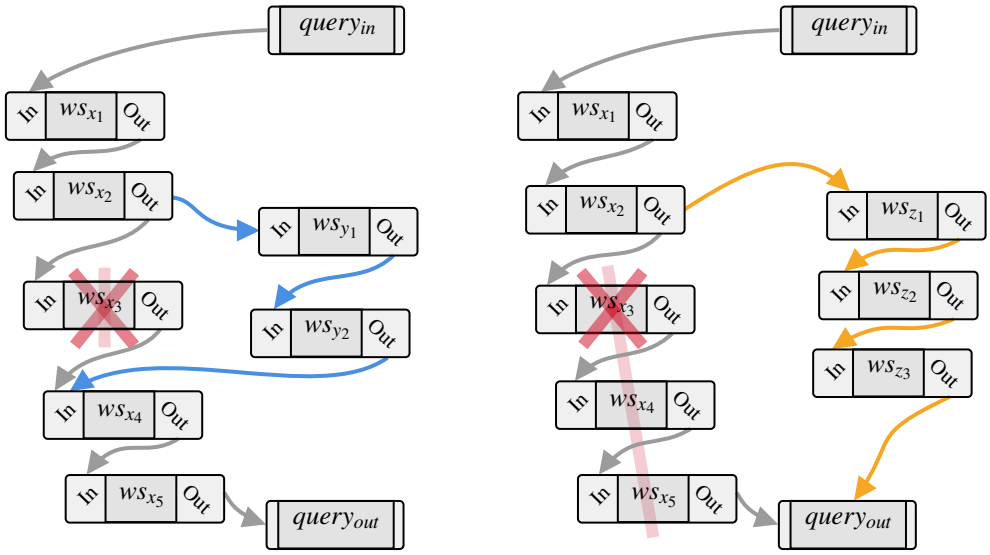

Figure 2: The two types of backups generated: replacing only one service on the left; and replacing all successive services or the "suffix", on the right.

example $\mathcal{X} = \langle ws_{x_1}, ws_{x_2}, ws_{x_3} ws_{x_4}, ws_{x_5} \rangle$. Edges are displayed only between consecutive services of the composition, but parameters can actually be chosen randomly from all outputs of previous services. Blue edges track parameters passing through sequence $\mathcal{Y} = \langle ws_{y_1}, ws_{y_2} \rangle$ that can replace ws_{x_3} in $\mathcal{X}$ - the first type of backup. Orange edges follow sequence $\mathcal{Z} = \langle ws_{z_1}, ws_{z_2}, ws_{z_3} \rangle$ that can replace $\langle ws_{x_3}, ws_{x_4} ws_{x_5} \rangle$ in $\mathcal{X}$ - the second type or "suffix" backup. If ws_{x_3} fails, either $\mathcal{Y}$ or $\mathcal{Z}$ could replace it. In Algorithm 2, inside FINDBACKUP() function the first backup type is obtained at line 17, and the second type at line 20. In the solution we prioritize for finding the first type of backup since it is more likely to have a shorter length, therefore being more efficient. The generator creates multiple of both of these backup types: first building paths of services and then assigning parameters accordingly.

Table 1 presents our evaluation metrics. First columns define the experiment instance size: the total number of distinct parameters, the total number of services in the initial repository and the total number of user queries. The following column specifies the total number of services used in all solutions found initially, for all queries. Out of these, we count how many have at least one possible backup: of the first or the second type. The last two columns include the running time consumed for solving all user requests and respectively for searching backups for each service used in compositions. Execution times are obtained for a Java implementation running on an `Intel(R) Core(TM) i5 CPU @2.40 GHz` machine with `8 GB RAM`.

For each query, one service is deleted, simulating its failure. This service can be either part of the initial composition found by the algorithm or of the backup built by the generator. Our test setting ensures that the deletions are not limited to backup services, therefore covering all possible cases: either replacing a composition service with a backup (the branch at line 6 in Algorithm 3) or just computing a new backup if a previous backup service fails (the branch at line 15 in Algorithm 3).

Finally, we also measured the naive reconstruction of all the compositions after deletion of services. These produced similar run times as the initially built solutions. This is expected since the problem instance is almost as large as in the initial setting.

Our measurements give an insight into the stress inflicted on the composition engine in practice. Two choices would be possible in case of failure: recompute a composition naively, which would have a similar cost with the initial computation of the composition, or using our pre-computed backups. Obviously, the backup option is more advantageous. The switch to the alternative composition is done instantly eliminating the overhead on computing a new solution. Even though the composition recomputation is not very high, it would still be desirable to avoid any disruption. Our solution practically eliminates any downtime caused by the failure. Moreover, for situations when no pre-computed backups are available, the user or service administrator can be warned ahead of time of critical services in the

Table 1: Experimental algorithm results. All run times are in seconds.

$\|\mathbb{P}\|$ parameters	$\|\mathbb{R}\|$ repository	$\|\mathbb{Q}\|$ queries	total services used	backups found of each type		build solutions	search backups
1000	100	5	23	15	2	0.005	0.022
1000	500	20	171	63	12	0.07	0.35
10000	1000	100	464	232	29	0.56	2.02
10000	2500	20	905	273	132	0.41	9.05

composition that are unrecoverable, and for which other measures could be taken (i.e., adding a new equivalent service in the repository). Finally, we observe that the time for building backups is not very significant, and does not add delays for the user since this operation is performed asynchronously after the initial composition is retrieved.

5 RELATED WORK

The solution described in (Cardinale and Rukoz, 2011) is probably one of the closest techniques to our work in providing fault tolerance in Web Service Composition. The approach is based on a Colored Petri Nets (CPN) modeling, used for backward recovery in the composition chain when a service fails, which essentially leads to a re-composition. However, the technique assumes several particular traits for the component services, such as being compensatable (i.e., some other service can semantically undo the execution of the failed service), leading to a transactional property for the composition. Our approach on re-composing relies on finding sequences of services providing the needed input and output parameters.

In (Vargas-Santiago et al., 2017), the authors survey solutions that use checkpointing mechanisms. Such solutions are mostly focused on situations when the failure of a service is not complete (i.e., the machine where the service resides is still operational). Checkpoints are used to mark a safe state of the service periodically. In case of a service malfunctioning, rollback to a specific checkpoint is possible, followed by resuming the service execution from such a safe state. Local recovery is defined in such a manner for individual services, while global recovery implies the rollback to a previous state for the entire system (i.e., covering the complete composition). Such approaches are different from our solution, typically not taking into account the impossibility of recovery of a failed service, and not considering finding an alternative composition. Also, the use of resources is higher in checkpointing mechanisms, requiring periodical state storage.

In (Laranjeiro and Vieira, 2008), the authors propose an architecture that relies on proxy service components. Each proxy service can invoke redundantly a series of alternate Web Services that provide the needed functionality in the composition. Therefore, when one or some of these services fail, another one accessible by the same proxy can replace it. The choice of the service is the outcome of a voting mechanism that takes into account evaluation metrics for low-cost redundancy. The solution is focused on the functionality aspects of the proposed architecture, such as integrating custom adapters that might be needed for interfacing with redundant services - e.g., adapting from a temperature scale to another. It does not discuss an effective algorithm for establishing the composition chain or dealing with the situation when a re-composition is needed following failures; i.e., when proxy services would fail, and an alternative composition would be required. The paper actually acknowledges that a consistency issue in case of service failures is subject of future work.

The solution in (Rao et al., 2007) discusses practical integration of WSC for achieving fault-tolerance, in particular referring to the integration of an external tool: a fault-tolerant planner - BIFROST - for addressing the issue of faults. The paper does not introduce a specific dedicated failover mechanism.

6 CONCLUSION

We approached in our solution the Web Service Composition problem from a highly practical perspective. We believe that specific failure recovery integration would be feasible in a real case scenario, where services can break at any time or can be deleted. We integrated service maintenance with the composition algorithm, in a manner that we consider an appropriate failover for situations when any used service becomes unresponsive. If an alternative composition path exists, our solution selects this as a backup, which can be used to restore the functionality instantly. The proposed resilience mechanism does not address only software faults as some of the previous work, but can

also work in a complete hardware failure scenario. In this initial work, we implemented our algorithm for the simplest version of service composition that matches parameters by names. The algorithm was evaluated on an automatically generated test suite that simulates complex scenarios.

Multiple additions to our solution can be considered for future work. First, new services could be dynamically added to the repository. Existing valid compositions could be optimized through shortening following such addition. Second, a more extensive evaluation could be conducted over an actual distributed set of services, simulating effective breakdown of nodes and their re-addition into the network. This would give more accurate insight into the delays caused by faults and the naive re-composition option, which we believe could be more severe than in our synthetic test setting. Other future work we consider is taking into account modern variations of the composition problem, like modeling semantics, enabling Quality of Service measures for services and generated compositions, or considering stateful services. We believe that our proposed solution offers a solid ground for pursuing such extensions.

REFERENCES

Blake, M. B., Tsui, K. C., and Wombacher, A. (2005). The EEE-05 challenge: A new web service discovery and composition competition. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 780–783.

Cardinale, Y. and Rukoz, M. (2011). Fault tolerant execution of transactional composite web services: An approach. In *Proceedings of The Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, UBICOMM 2011.

Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., et al. (2001). Web services description language (WSDL) 1.1.

Diac, P. (2017). Engineering polynomial-time solutions for automatic web service composition. *Procedia Computer Science*, 112:643–652.

Laranjeiro, N. and Vieira, M. (2008). Deploying fault tolerant web service compositions. *International Journal of Computer Systems Science and Engineering*, 23:337–348.

MoreThanCoding.com (2012). The web apis you use will fail.

Rao, D., Jiang, Z., and Jiang, Y. (2007). Fault tolerant web services composition as planning. In *Proceedings of The International Conference on Intelligent Systems and Knowledge Engineering 2007*. Atlantis Press.

Ţucăr, L. and Diac, P. (2018). Semantic web service composition based on graph search. *Procedia Computer Science*, 126:116–125.

Vargas-Santiago, M., Pomares-Hernandez, S., Morales Rosales, L. A., and Hadj-Kacem, H. (2017). Survey on web services fault tolerance approaches based on checkpointing mechanisms. *Journal of Software*, 12:507–525.

Weise, T., Bleul, S., Comes, D., and Geihs, K. (2008). Different approaches to semantic web service composition. In *Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 90–96.