Self-healing of web service compositions: a specification rewriting approach

Rafael Ferreira Toledo, Umberto Souza Da Costa, Martin Musicante, Genoveva Vargas-Solar

Self-healing of web service compositions: a specification rewriting approach

Rafael Ferreira Toledo

Cheriton School of Computer Science,
University of Waterloo,
Waterloo, Canada
Email: rftoledo@uwaterloo.ca

Umberto Souza da Costa* and
Martin A. Musicante

Computer Science Department (DIMAp),
Federal University of Rio Grande do Norte,
Natal, Brazil
Email: umberto@dimap.ufrn.br
Email: mam@dimap.ufrn.br
*Corresponding author

Genoveva Vargas-Solar

LIG-LAFMIA,
CNRS,
Grenoble INP,
University Grenoble Alpes,700 avenue Centrale,
Domaine Universitaire, 38401 Saint-Martin-d'Hères, France
Email: genoveva.vargas@imag.fr

Abstract: We present an approach that improves the robustness of web service compositions enabling their recovery from failures that can happen at different execution times. We first present a taxonomy of failures as an overview of previous research works on the topic of fault recovery of service compositions. The resulting classification is used to propose our self-healing method for web service compositions. The proposed method, based on the refinement process of compositions, takes user preferences into account to generate the best possible recovering compositions. In order to validate our approach, we produced a prototype implementation capable of simulating and analysing different scenarios of faults. Our work introduces algorithms for generating synthetic compositions and web services. In this setting, the recovery time, the user preference degradation and the impact of different locations of failure are investigated under different strategies, namely local, partial or total recovery. These strategies represent different levels of intervention on the composition.

Keywords: web services; self-healing; user preferences; service composition rewriting; fault handling; service composition maintenance; adaptive error recovery; service replacement; specification requirements; service composition specification.

Reference to this paper should be made as follows: Toledo, R.F., da Costa, U.S., Musicante, M.A. and Vargas-Solar, G. (2020) 'Self-healing of web service compositions: a specification rewriting approach', *Int. J. Web and Grid Services*, Vol. 16, No. 2, pp.172–199.

Biographical notes: Rafael Ferreira Toledo received his BS in Control and Automation Engineering from the Federal Fluminense Institute in 2016. He obtained his MS in Systems and Computation from the Federal University of Rio Grande do Norte in 2018. He is currently pursuing his PhD in Computer Science from the University of Waterloo. His academic interests include service-oriented computing, features interaction, and programming languages.

Umberto Souza da Costa received his BS in Computer Science from the Federal University of Rio Grande do Norte in 1998. He obtained his MS in Systems and Computation from the Federal University of Rio Grande do Norte in 2000. He received his PhD in Computer Science from the University Federal de Minas Gerais in 2005. He is currently an Associate Professor in the Department of Informatics and Applied Mathematics (DIMAp) at the Federal University of Rio Grande do Norte. His academic interests include programming languages, service-oriented computing and cloud computing.

Martin A. Musicante received his BSc at the ESLAI, Argentina, in 1988 and MSc and PhD in Computer Science at the Universidade Federal de Pernambuco, Brazil, in 1990 and 1996, respectively. He is a Professor at the Universidade Federal do Rio Grande do Norte-UFRN in Natal, Brazil. He is part of the graduate program in Computer Science at the UFRN. He was an Associate Researcher at the LI – Université François Rabelais Tours in 2002–2011 and at the LIFO – Université d'Orléans, France, since 2008.

Genoveva Vargas-Solar received her first PhD on Computer Science from the University Joseph Fourier and her second PhD from the University Stendhal. She obtained her Habilitation à Diriger des Recherches (HDR – tenure) from the University of Grenoble. Her research interests in computer science concern distributed and heterogeneous databases, reflexive systems and service-based database systems. She contributes to the construction of service-based database management systems.

1 Introduction

In the past decades, much effort has been driven to evolve the technology of web service composition and to increase the quality delivered by systems designed following the service-oriented architecture (SOA) paradigm. As for any distributed system, it becomes hard to guarantee the desired behaviour of a web service composition (Sheng et al., 2014). The ability to manage the execution of web service compositions may improve the reliability and fault-tolerance of the deployed system. Devising mechanisms to recover from defects opens considerable research opportunities (Yu et al., 2008; Papazoglou et al., 2007; Sheng et al., 2014).

Web service compositions operate on dynamic environments, prone to the occurrence of unpredictable disruptions and changes that can affect the execution of the system. These problems can appear at different levels of the service composition stack, ranging from defects at individual services to defects at the underlying infrastructure. The

availability of the remotely located services, the quality attributes delivered by the services, and the operating condition of the network are intrinsic to SOA and may be the source of failures. According to Papazoglou et al. (2007), providing services capable of detecting and correcting faults, while preserving the runtime performance with a minimal dependency of human interaction is a research challenge. Also, failure recovery is crucial for the proper and effective delivery of web service functionalities (Yu et al., 2008). The temporary unavailability of a vital component service and changes of the quality attributes offered by a service are examples of failures.

Classifying faults in the context of service compositions is a step towards the definition of more robust SOA applications. Several classifications exist, adopting different criteria, including functional and non-functional aspects, as well as different granularities of the composition (Fugini and Mussi, 2006; Bruning et al., 2007; Chan et al., 2007; Erradi et al., 2006; Liu et al., 2010; Wang et al., 2009; Yu et al., 2008). Most of these works propose the use of their taxonomy to define self-healing mechanisms for web service compositions. In this paper, we adapt the taxonomy proposed by Fugini and Mussi (2006) to define a novel self-healing mechanism, based on the composition refinement method presented in Ba et al. (2016).

This paper describes the following contributions:

- an overview of research works improving the robustness of web service compositions
- a taxonomy characterising failures and classifying them according to the location of the fault in the web composition stack
- the use of the proposed taxonomy to define a self-healing approach for web service compositions
- a prototype implementing our recovery approach, as well as the generation of synthetic compositions and web services, used to validate our proposal.

The remainder of the paper is organised as follows: Section 2 introduces some basic notions about SOA, service composition rewriting, as well as an overview of failure recovery approaches. Section 3 looks into related work. Section 4 presents a taxonomy of failures on service compositions. Section 5 introduces the main contribution of this paper, namely a self-healing mechanism for web service compositions based on a taxonomy of failures classifying them according to their location in the web composition stack. Our proposal validation is described in Section 6. Section 7 discusses the results of our work, as well as research opportunities for future work.

2 Basic concepts

This section gives an overview of the concepts behind web service composition upon which we defined our proposal.

2.1 Notions of SOA

SOA (Papazoglou et al., 2007) provides a way of designing software applications composed by services. Those services may be end-user or distributed applications

accessible via published and discoverable interfaces. According to Papazoglou et al. (2007), SOA can be viewed as key to fulfil the visionary promise of service oriented computing (SOC), where applications are built over loosely-coupled components, to create dynamic business processes and promote the agile development of software.

As pointed out by Raines (2009), web service standards implement general SOA concepts. These standards have become practical tools used by enterprise engineers for SOA projects. IBM defines a web service as a breed of web applications that are self-contained, self-describing, modular and can be published, located and invoked across the web (Sheng et al., 2014; Shah and Patel, 2016).

The SOA architecture is based on three elements: the *service provider*, the *service registry* and the *service requester*. The service provider publishes and implements the web service. The service requester invokes the service after retrieving its description from a service registry or repository (like UDDI), where the available web services are advertised and located (Sheng et al., 2014; Papazoglou et al., 2007; Sheng et al., 2006). Services with similar functionality may be implemented by different providers. Those services can appear at any repository and have distinct associated non-functional attributes.

Atomic web services are applications that do not depend on another web service to respond to the service requesters. The composition of web services is the combination of services, to implement a set of functionalities of a more complex business process. Some concerns are directly related to the development of web service compositions, such as component access, conversation management, control flow, data flow and data transformation (Lemos et al., 2016).

The life cycle of a service composition involves four phases (Sheng et al., 2014):

- *definition:* the web service composition is specified by an abstract model, including service requirements and preferences
- *service selection:* an available concrete service is selected to fulfil the requirements of each service specified by the abstract model
- *deployment:* the selected services are integrated, so that the composition is deployed as an executable service
- *execution:* the composition is executed together with monitoring and fault-handling mechanisms.

These phases describe the process of combining concrete services to implement the desired behaviour of the application (Orriëns et al., 2003). In a previous work (Ba et al., 2016), we introduced a refinement algorithm to be used in the service selection phase. This algorithm provides the basis for our self-healing approach.

2.2 Service composition refinement

Query rewriting techniques have been applied address the automatic selection and composition of web services (Barhamgi et al., 2010; Mesmoudi et al., 2011; Ba et al., 2016; Costa et al., 2013; Zhao et al., 2012]. The automatic composition of web services may consider two main phases: the selection of available services to integrate the final composition, and the configuration of the selected components to build the application. The service composition rewriting must be able to identify the services that cover the

expected functionalities of the composition and generate a concrete composition consisting of the identified services.

In Costa et al. (2013) and Ba et al. (2016) we propose and use an adapted version of the Minicon algorithm (Pottinger and Halevy, 2001) to produce concrete compositions from their abstract specification. In order to illustrate the method, let us consider a running example that defines a simple travel agency service. This example will be used throughout the paper.

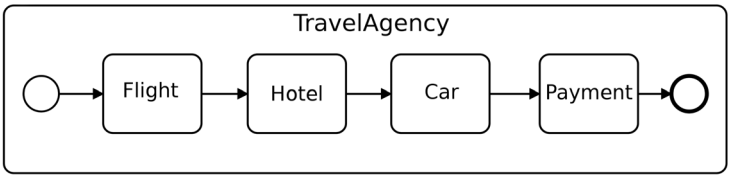
Example 1 (composition specification): the *TravelAgency* service is given by the following abstract specification:

$$\begin{aligned} TravelAgency(org, dst, dep, ret, rcpt) \equiv\ & Flight(org, dst, dep, ret, flnv) \\ & Hotel(flnv, hlnv), \\ & Car(hlnv, clnv), \\ & Payment(flnv, hlnv, clnv, rcpt) \end{aligned}$$

The *TravelAgency* service provides functionalities for booking a flight (given the departure and arrival places and dates), booking hotel and car and processing the payment. Each of these functionalities is represented by a predicate. The specification is defined by the predicates *Flight*, *Hotel*, *Car* and *Payment*. Figure 1 shows the specification of the TravelAgency composed service in a BPMN diagram (Wohed et al., 2006).

■

Figure 1 BPMN diagram for the TravelAgency specification



Each predicate receives input data, such as preferred dates and flight for the trip. The variables fInv, hInv and cInv represent the invoices returned by each booking-related functionality. The service payment uses these invoices to calculate the due cost for the trip and perform the payment using the information on those invoices. The parameter rcpt is the overall receipt and indicates to the user the success of the booking operation. The workflow defined by the composition is implicitly defined by the data dependencies among functionalities.

Each functionality is intended to be provided (or covered) by one or more web services. In SOA, services are expected to be indexed through a service registry. We assume that services are also specified by predicates, defining their functionalities.

Example 2 (service specifications): in our example, we may have the following web services, specified using the same functionalities as the composition:

$$Gol(org, dst, dep, ret, flnv) \equiv Flight(org, dst, dep, ret, flnv)$$

$$Latam(org, dst, dep, ret, flnv) \equiv Flight(org, dst, dep, ret, flnv)$$

$$Booking(org, dst, dep, ret, flnv, hlnv, clnv) \equiv Flight(org, dst, dep, ret, flnv), Hotel(flnv, hlnv), Car(hlnv, clnv)$$

$$Ibis(flnv, hlnv) \equiv Hotel(flnv, hlnv)$$

$$Localiza(hlnv, clnv) \equiv Car(hlnv, clnv)$$

$$Expedia(flnv, hlnv, clnv) \equiv Hotel(flnv, hlnv) Car(hlnv, clnv)$$

$$Visa(flnv, hlnv, clnv, rcpt) \equiv Payment(flnv, hlnv, clnv, rcpt)$$

The services *Gol*, *Latam* and *Booking* provide the *Flight* functionality. Notice that *Booking* also implements the *Hotel* and *Car* functionalities.

■

Services may have different providers, and different quality parameters. The composition developer specifies their preferences during the refinement process by assigning a score to each concrete service available. The preference for each service is represented by a real number between zero (lowest preference) and one (highest preference). User preferences are used as selection criteria to favour the choice of services to be part of an actual composition. Preference values are used to represent any criteria for choosing an actual web service. Services with higher scores are preferred to those with lower scores.

We assume that preference values are part of the input to our rewriting process. The definition of preference values for concrete services is outside the scope of this work.

Example 3 (user preferences): in our travel agency scenario, the preference values associated to each service are: *Gol* (0.9), *Booking* (0.9), *Latam* (0.8), *Ibis* (0.9), *Localiza* (0.9), *Expedia* (0.7), *Visa* (0.9).

■

During the service selection phase, the rewriting algorithm splits the abstract composition specification into blocks and looks for web services providing the functionality of each block. The result of this phase is a set of *partial coverage descriptors* (PCDs). Each PCD contains information on how to use a web service to be part of the concrete composition. This information includes variable mappings (to bind variables in the specification to the arguments of the services), as well as, the name of the predicates covered by the service. Table 1 summarises the PCDs for our running example. Notice that the preference of the service defines the preference of the PCD.

The integration phase of the composition algorithm combines PCDs to produce actual compositions (Costa et al., 2013). The resulting compositions must comply with two restrictions:

1 to provide all the required functionalities

2 each functionality must be provided once in the composition.

Notice that one concrete service may cover more than one functionality of the specification (this is the case of PCD_2 and PCD_6 in Table 1).

Table 1 PCDs based on concrete services

PCD	*Service*	*Coverage*	*Preference*
PCD_1	Gol	Flight	0.9
PCD_2	Booking	Flight, hotel, car	0.9
PCD_3	Latam	Flight	0.8
PCD_4	Ibis	Hotel	0.9
PCD_5	Localiza	Car	0.9
PCD_6	Expedia	Hotel, car	0.7
PCD_7	Visa	Payment	0.9

The integration phase may produce several compositions. The POTI algorithm in Ba et al. (2016) uses user preference values to classify the resulting compositions, using the Pareto order (Kießling, 2002).

Example 4 (service composition): in the travel agency example, one of the most preferable compositions is formed as:

$$TravelAgency(org, dst, dep, ret, rcpt) \equiv Booking(org, dst, dep, ret, flnv, hlnv, clnv)$$
$$Visa(flnv, hlnv, clnv, rcpt)$$

■

The POTI algorithm is done in two steps:

- *production of a set of PCDs:* given an abstract specification $A = \{f_1, \ldots, f_k\}$, where each f_i stands for a functionality, and a set $\mathcal{S}$ of web services available in the registry, this step generates the set of all PCDs that can be used to cover functionalities of A
- *combination of PCDs:* this step selects PCDs from the previous phase and combines them produce actual compositions.

For our running example, given the composition specified in Example 1 and the set of web services of Example 2, the first phase produces the PCDs in Table 1. Notice that one PCD may cover more than one functionality.

During the combination phase, the rewriting process checks some constraints before integrating PCDs. These constraints include:

1 all functionalities of the specification needs to be covered

2 each functionality must be covered by just one PCD

3 the use of parameters on the predicates must be consistent.

For the sake of simplicity, the verification of these constraints are abstractly represented as a call of a function *compatible*, which take a set of PCDs as input and returns a Boolean value. The computational details of this function can be found in Costa et al. (2013). Once a set of compatible PCDs is found, it is integrated into a concrete composition. This integration is represented here by the function *combine*.

Several compositions may be produced by the method. User preferences are used to chose the most suitable rewritings. For the running example, the result of the second phase is given by the service composition of Example 4.

In this paper we adapt the algorithm POTI to provide alternative services in case of failure of parts of a composition. Specifically, our method explores the PCDs generated at the selection phase of POTI to replace parts of the failed composition, while preserving the overall functionality.

2.3 Self-healing of web services

Faults, errors and failures are some of the terms used to define elements that affect the well-functioning of a given system. According to Avizienis et al. (2001), a *failure* is the deviation of the service from the correct behaviour. This unexpected behaviour is the consequence of the occurrence of *errors* that alter the service, being noticed by the user when the behaviour reaches the service interface. *Faults* are the possible causes of an error.

Fault-tolerance is the preservation of correctness of a system in the presence of faults (Avizienis et al., 2001). Fault-tolerance is fundamental to self-healing, which is the ability of a system to discover, diagnose and react to faults without disrupting the runtime environment. A fault-tolerant mechanism is typically implemented by error detection and subsequent system recovery (Avizienis et al., 2001). A fault taxonomy for SOA helps refining possible reactions for runtime faults, guiding fault injection tests executed on services during development, and consequently improving robustness, reliability and availability of SOA components (Bruning et al., 2007).

Recovery actions can be classified as service-oriented actions and data quality recovery actions (Fugini and Mussi, 2006). The former deals with invocation, orchestration and choreography aspects of web services. Data quality recovery actions can be achieved by methods like data cleaning by manual identification, multiple source identification and verification of data specifications.

The application of recovery actions to correct the errors detected during execution contributes to achieving fault-tolerance in a web service composition. Basic failure reactions for web service composition are (Erradi et al., 2006; Wang et al., 2009; Liu, A., Li et al., 2010; Fugini and Mussi, 2006):

- *Notify:* the system signals the occurrence of a failure, possibly adding entries to a log file. This reaction can be triggered by failures at any level of the taxonomy.
- *Ignore:* the system does not actively interfere with the execution of the service. This reaction applies to failures that do not affect the main goal of the composition.
- *Retry:* applies to transient failures due to instabilities of hardware or software. This action may be only offered for services that can be executed multiple times without affecting the consistency of the process. This reaction may consider deadlines such as a maximum number of tries or a time limit. *Retry* is suitable for failures on functional behaviour or mismatch on the output of the service.
- *Replace:* the failed service is substituted with another. The new service is expected to be equivalent in terms of functionality and QoS. This reaction is usually triggered after the unavailability of a service or its inability to succeed. Replace strategies

depend on mechanisms to identify available compatible services to be integrated in the composition. The failed service may be replaced with an atomic or composed service.

- *Recompose:* this action is adopted in the case of failure of all services in the composition. Recomposing a composition consists on establishing an alternative business process with the same primary goals of the failed composition. The execution of this reaction works as a *replace* action applied to all web services components.

Both replace and recompose make use of compensation services, or rollbacks, to compensate partially executed faulty processes. They are usually considered pre-conditions of the recovery strategies in those cases (Wang et al., 2009).

Self-adaptation methods for service compositions that can implement the listed reactions are classified as policy-based and replacement-based approaches (Yuan et al., 2018). The former is defined as methods that follow policies defined during design time and cannot deal with context events beyond the policies. While, in case of unexpected changes, the replacement-based approaches substitute services by new selected ones according to the original specification. In Wang and Yang (2018) for example, propose the concept of dynamic Petri nets to describe fault-handling strategies for service compositions, considering dynamic replacement of transitions. The optimisation of the service matching process, as the one presented in the current work, is stated as a challenge for the implementation of the fault-handling strategies.

3 Related work

Many approaches have been proposed for the categorisation of faults in SOA systems. In Wang et al. (2009) define a fault taxonomy aiming to deal with business constraints violations, including business and technical faults. Based on the taxonomy, the authors provide an instrumentation template for constraint violation and runtime fault handling of business process execution language (BPEL) processes. Two approaches are considered:

1 a basic replacement plan that replace services according to a static list provided by the user

2 a more elaborated strategy that uses a service repository to discover possible replacing candidates.

That repository saves composed services, annotated with fault ratio. Those services can be retrieved and selected for recomposition.

The work in Bruning et al. (2007) defines a taxonomy for SOA based on the life cycle of the service composition. The taxonomy contributes to the methodology of using fault injections to test fault recovery in service compositions. The authors' goal is to cover as many fault classes as possible while minimising the number of test cases. The categorisation of faults facilitates the testing approach by abstracting from concrete implementations. The taxonomy associates system development phases to faults at the higher-level. These levels are refined into atomic fault cases. The proposal does not categorise, relate or formulate runtime fault-handling strategies.

In Liu et al. (2010) present a framework for the fault-tolerant composition of transactional web services. The framework includes a fault-tolerant mechanism that combines exception handling and transaction techniques. This combination is based on the identification of fault models, a set of high-level exception handling strategies, a new taxonomy of transactional web services and critical features, including service transfer and vitality degree. The ultimate goal of the framework is to build an integrated environment for the specification, verification, and execution of fault-tolerant service compositions. The strategy for replacing faulty services is statically defined: the approach does not propose the automatic discovery of replacing services.

The taxonomy in Chan et al. (2007) is used to contribute to dynamo (Baresi et al., 2007), a toolset that proposes runtime monitoring and recovery strategies for BPEL. Dynamo uses a monitor to assess the quality of each service during runtime. Once a service fails, the system looks for a backup service. A backup service may be indicated by the user. Alternatively, the monitor may choose a replacement based on the quality assessment of previously used services.

The work in Fugini and Mussi (2006) proposes a self-healing approach for web services and present a classification of faults. This categorisation is distributed in three different levels, namely *web service*, *application* and *infrastructure*. Their work proposes a self-healing platform that performs a semantic-based analysis to compare faulty services with candidates for substitution. Human intervention might be required when services have different signatures.

The works cited above propose tools for web service composition that enable the monitoring of faults, the specification of recovery strategies and the execution of remedial actions. Most of these works consider the replacement of a faulty service as a reaction to failures, by replacing individual services. Substitute services are usually provided by the user that explicitly defines the candidate for replacement. Some solutions query a service registry on-the-fly. Some authors mention the use of semantic analysis to identify replacement services, without providing details about their approach. Differently from our proposal, these works do not consider the possibility of substituting the faulty service with a composition of services.

4 Fault taxonomy

We present a taxonomy of failures for web services. Our taxonomy results from the analysis of proposals in Chan et al. (2007), Bruning et al. (2007), Li et al. (2014), Wang et al. (2009), Fugini and Mussi (2006), Liu et al. (2010) and Simmonds et al. (2013). Similarly to Fugini and Mussi (2006) and Wang et al. (2009), failures are classified into three main levels: *service*, *composition* and *infrastructure*.

The failures listed in this work are studied in the literature, and they are distributed in a way that contributes to the discussion of failure recovery of service compositions. Table 2 summarises our classification.

Each level of failure has a set of conditions that should monitored by the runtime environment. The definition of levels in which the failure may arise helps to choose a recovery strategy. Recovery actions for each case can be executed by a recovery system.

Table 2 Fault taxonomy

Level	*Violation*	*Example*
Component	Content	Incorrect results
		Service provided different from expected
	Timing	Time-out
Composition	Quality of service	Low availability
		High rate of error
		Financial faults
		SLA violation
	Compatibility	Missing parameter
		Mismatch data types
		Incorrect order
	Coverage	Missing parts of the composition
Infrastructure	Platform	Server crashed
	Network	Missing connection
		Low bandwidth

4.1 Service level

This level is specifically related to aspects of each component service of the web service composition. This context may consider both the quality and the correctness of the service. Failures at this level are classified into *content* and *timing* (Wang et al., 2009; Chan et al., 2007; Fugini and Mussi, 2006; Li et al., 2014).

Content violations are related to the definition and expected outputs of a web service. Content failures occur when there is a mismatch between the expected and the actual data produced by the service. Examples of failures in this category are the delivery of incorrect results from the service and the provision of a service different from expected. The former case results into the incoherent behaviour of the service and may compromise the final result of the composition. In the case where a wrong service is provided, the service output may be functionally compliant, but deviating from other aspects of the specification. This kind of problem may happen due to an incorrect service selection before the deployment of the composition.

Timing failures are related with timeouts and other errors that may affect the time of arrival and delivery of data for the service that may impact the non-functional specifications of the web service composition (Chan et al., 2007).

4.2 Composition level

The composition level concerns the conversation between component services, as well as the compliance of the composition with regard to its specification (Fugini and Mussi, 2006; Bruning et al., 2007; Wang et al., 2009). The requirements of a composition include functional and non-functional aspects, established during the definition phase. The concrete composition is expected to meet those requirements at runtime. In this way,

failures at the composition level include *quality of service (QoS)*, *compatibility* and *coverage* failures.

Compatibility failures are related to the mismatch of exchanged data between services. These failures may occur due to differences in the arguments or protocols considered by the services during the exchange of data (Wang et al., 2009). Examples of failures in this class are missing parameters or incorrect data types.

Coverage failures are defined as violations of the specification regarding the binding between the expected functionalities and the component services.

QoS failures are related to violations of non-functional requirements of the composition, such as those concerning availability, response time, throughput, security and price. These attributes and their expected values can be formalised in a service level agreement (Bianco et al., 2008).

4.3 Infrastructure level

This level focuses on failures at the runtime environment supporting the composition (Wang et al., 2009; Liu et al., 2010). Failures at this level impact the execution of the web services due to infrastructure problems. This category includes *platform* and *network* failures.

Platform failures occur when a service is unavailable due to a problem in the client or in the device providing the service (Wang et al., 2009). Network failures are due to communication errors between services and clients, such as connectivity losses or low bandwidth. Both platform and network violations may cause unavailability of web service components and a consequent failure of the composition (Chan et al., 2007).

5 Proposed self-healing approach

We propose a self-healing mechanism for web service compositions. The proposal is intended to be part of a platform that:

1 identify runtime failures

2 propose reactions to the failures

3 recover the system.

We focus on the second item above, by introducing an algorithm to find most preferable compensation services to replace portions of the composition. Our algorithm is based on the composition refinement method POTI (Ba et al., 2016) for the orchestration of web services.

Given a composition obtained as the result of POTI, the problem of replacing a failed service can be addressed by considering another rewriting, that complies to the original specification, such that the new composition replaces the failed service by one of the next preferable candidates. The collection of user preferences associated to the web services represents a particular challenge which is beyond the scope of the current work. In Tian et al. (2019), for example, address some of the difficulties of recommending services to an user with unknown explicit preferences. In the general context of recommendation systems, Parra et al. (2011) propose a method for mapping implicit

feedback of users, like the popularity of a given element, to explicit numerical user ratings.

For a composition consisting of services S_1, …, S_n, our algorithm considers three incremental recovery levels: *local*, *partial* and *total*. Given that a service S_i fails, each recovery level defines which part of the composition must be replaced. At the local level, the algorithm tries to replace just the failed service S_i. If it is not possible to recover locally, the algorithm steps to the partial level of recovery, by replacing S_i and the subsequent services. This occurs when there is no possible substitution for the individual service. In this case, the algorithm tries to replace the sub-composition defined by S_i, …, S_n. If there is no possible replacement at the partial level, the algorithm tries to obtain a rewriting for the whole composition.

Local recovery

It occurs when a single service of the composition is to be replaced. This service can cover one or more functionalities. These functionalities may be covered by one or more other services. In Example 5 we consider the case where the failed service covers more than one functionality.

Example 5 (local recovery): consider the composition from Example 4:

$$\begin{aligned} \mathit{TravelAgency}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{rcpt}) \equiv\ & \mathit{Booking}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{flnv}, \mathit{hlnv}, \mathit{clnv}) \\ & \mathit{Visa}(\mathit{flnv}, \mathit{hlnv}, \mathit{clnv}, \mathit{rcpt}) \end{aligned}$$

In this composition, the service *Booking* covers the functionalities *Flight*, *Hotel* and *Car*. *Visa* covers the *Payment* functionality. In the case of failure of *Booking*, our algorithm tries to perform a local recovery to cover each of the functionalities of *Booking* with another service. Our algorithm considers the user preference of each service to obtain a new composition which does not include the failed service. In the case of the travel agency, the preferences defined in Example 3 are used to obtain the composition:

$$\begin{aligned} \mathit{TravelAgency}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{rcpt}) \equiv\ & \mathit{Gol}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{flnv}) \\ & \mathit{Ibis}(\mathit{flnv}, \mathit{hlnv}), \\ & \mathit{Localiza}(\mathit{hlnv}, \mathit{clnv}), \\ & \mathit{Visa}(\mathit{flnv}, \mathit{hlnv}, \mathit{clnv}, \mathit{rcpt}) \end{aligned}$$

Notice that *Booking* was locally replaced with three services that, combined, provides the same set of functionalities. These three services have the same combined preference as *Booking*. Since we are in the context of a local recovery, the service *Visa* was not replaced.

■

Partial recovery

This level of recovery is tried by the algorithm when the local recovery does not succeed. In this case, not only the faulty service S_i is replaced, but all the subsequent services up to S_n.

Example 6 (partial recovery): let us consider the composition produced by the recovery process in Example 5. Suppose that the service *Ibis* fails. In this case, using the services described in Example 2, there is no possible local recovery, since there is no other service covering just the *Hotel* functionality. In this situation, we look for services covering the part of the composition that has not been executed yet. Notice that, except for the failed service, the only way to cover the *Hotel* functionality is by using the service *Expedia*, that also covers *Car*. The partial recovery produces the following composition:

$$\begin{aligned} \mathit{TravelAgency}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{rcpt}) \equiv \mathit{Gol}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{flnv}) \\ \mathit{Expedia}(\mathit{flnv}, \mathit{hlnv}, \mathit{clnv}), \\ \mathit{Visa}(\mathit{flnv}, \mathit{hlnv}, \mathit{clnv}, \mathit{rcpt}) \end{aligned}$$

The service *Visa* is part of the solution because it did not fail and the rewriting process chooses it to cover the *Payment* functionality.

■

Total recovery

When it is not possible to perform a partial recovery of the composition, our algorithm will try to find a replacement for the whole composition. Some functionalities may be re-executed.

Example 7 (total recovery): again, consider the composition resulting from Example 5, but at this time, we assume that the failed service is *Localiza*, that covers the *Car* functionality.

The alternative services covering *Car* are *Booking* and *Expedia* (Example 2). Since these alternative services also cover functionalities already executed, nor the local or partial recovery will succeed. The total recovery produces the following composition:

$$\begin{aligned} \mathit{TravelAgency}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{rcpt}) \equiv \mathit{Booking}(\mathit{org}, \mathit{dst}, \mathit{dep}, \mathit{ret}, \mathit{flnv}, \mathit{hlnv}, \mathit{clnv}) \\ \mathit{Visa}(\mathit{flnv}, \mathit{hlnv}, \mathit{clnv}, \mathit{rcpt}) \end{aligned}$$

■

In our recovery algorithm, user preferences guide the selection of alternative services. Notice that due to the way in which replacement services are chosen, there is no degradation in terms of functionality in the composition after its recovery. Our method is based on the selection phase of POTI for choosing PDCs (see Section 2.2), thus preserving the functionality of the composition. Recall that PCDs represent sets of services that can be used as candidates for replacement.

Our method preserves the functionality of the composition but it may cause degradation of the composition in terms of user preference. This is a consequence of choosing the replacement solutions by using the Pareto ordering for preference values. Notice that preference values may be used to express any criteria, such as time or space consumption, subjective preference, etc. In this way, the decrease of preference may indicate the degradation of any non-functional concern of the composition.

5.1 The self-healing approach

Our method encompasses the composition rewriting process described in Ba et al. (2016). We suppose that the POTI algorithm is used to generate compositions for a given abstract specification. In this process, the set $\mathcal{PCD}$ of PCDs is produced. In our self-healing scenario, we consider that the most preferable composition C is the one initially deployed, being formed by the set of services $\{S_1, \ldots, S_n\}$.

Algorithm 1 Self-healing

Input:

- The set $\mathcal{PCD}$ of all available PCDs.
- The running composition $C \equiv \{S_1, \ldots, S_n\}$.
- The failed service S_i.

Output:

- The recovering composition R.

1: **function** Heal($\mathcal{PCD}$, C, S_i)
2: $\mathcal{PCD}_C \leftarrow$ PCDsOf(C)
3: $\mathcal{F}_L \leftarrow Funct(\mathcal{PCD}[S_i] \cap \mathcal{PCD}_C)$
4: $\mathcal{F}_P \leftarrow Funct(\mathcal{PCD}[S_i, ..., S_n] \cap \mathcal{PCD}_C)$
5: $\mathcal{F}_T \leftarrow Funct(\mathcal{PCD}_C)$
6: $R \leftarrow$ Recover($\mathcal{PCD}$, $\mathcal{PCD}_C$, S_i, $\mathcal{F}_L$)
7: **if** R is the empty composition **then**
8: $R \leftarrow$ Recover($\mathcal{PCD}$, $\mathcal{PCD}_C$, S_i, $\mathcal{F}_P$)
9: **end if**
10: **if** R is the empty composition **then**
11: $R \leftarrow$ Recover($\mathcal{PCD}$, $\mathcal{PCD}_C$, S_i, $\mathcal{F}_T$)
12: **end if**
13: **return** R
14: **end function**

Algorithm 1 describes our recovering approach. The algorithm receives the set $\mathcal{PCD}$ containing the PCDs produced by POTI, the running composition C formed by the set of services $\{S_1, \ldots, S_n\}$, as well as the identification of the failed service S_i. The subset of $\mathcal{PCD}$ that covers the composition C is denoted by $\mathcal{PCD}_C$ (line 2). The $\mathcal{F}_L$, $\mathcal{F}_P$ and $\mathcal{F}_T$ sets (lines 3 to 5) contain the functionalities to be covered at local, partial and total recovery levels, respectively. The algorithm first tries a local recovery (line 6). If the local recovery does not succeed, the algorithm tries a partial recovery (line 8). Finally, if the partial recovery is not possible, a total one is tried (line 11). The algorithm returns a non-empty recovering composition R whenever a solution is found.

Algorithm 2 produces new service compositions. It takes:

1 the set (registry) $\mathcal{PCD}$ of all available PCDs

2 the set $\mathcal{PCD}_C$ of PCDs used in the original composition C

3 the identification of the failed service S_i

4 the set $\mathcal{F}$ of functionalities covered by services to be replaced in the recovering composition.

Algorithm 2 Recover

Input:

- The set $\mathcal{PCD}$ of all available PCDs.
- The set $\mathcal{PCD}_C$ of PCDs used in C.
- The failed service S_i.
- The set $\mathcal{F}$ of functionalities to be covered.

Output:

- The recovering composition.

1: **function** Recover $(\mathcal{PCD}, \mathcal{PCD}_C, S_i, \mathcal{F})$

2: $\mathcal{PCD}_F \leftarrow \mathcal{PCD}_C \setminus \{p \in \mathcal{PCD}_C \mid p \text{ covers } f \in \mathcal{F}\}$

3: $\mathcal{PCD}_H \leftarrow \mathcal{PCD} \setminus \mathcal{PCD}[S_i]$

4: $\mathcal{PCD}_U \leftarrow \{p \in \mathcal{PCD}_H \mid p \text{ covers } f \in \mathcal{F}\}$

5: **for** each $\mathcal{P} = \{P_1, ..., P_m\} \subseteq \mathcal{PCD}_c$

6: such that

7: (i) $\mathcal{F} \subseteq Funct\{\mathcal{P}\}$ and

8: (ii) $\forall r \neq s.\, Funct(P_r) \cap Funct(P_s) = \emptyset$

9: **do**

10: **if** $Compatible(\mathcal{P} \cup \mathcal{PCD}_F)$ **then**

11: **return** $Combine(\mathcal{P} \cup \mathcal{PCD}_F)$

12: **end if**

13: **end for**

14: **return** empty;

15: **end function**

This algorithm defines three sets of PCDs:

1 The set $\mathcal{PCD}_F$ of fixed PCDs (line 2). This set consists of all the PCDs of the original composition that will be maintained in the resulting composition. Notice that the contents of this set depends on the recovery level. This set is empty in total recoveries.

2 The set $\mathcal{PCD}_H$ of healthy PCDs (line 3), containing all the available PCDs, except those formed for the failed service S_i.

3 The set $\mathcal{PCD}_U$ of usable PCDs (line 4), containing the healthy PCDs that can be used to cover the functionalities in $\mathcal{F}$.

The loop in line 5, iterates over those sets $\mathcal{P}$ of PCDs that may be used to recover the composition. Conditions at lines 7 and 8 state that each functionality in $\mathcal{F}$ must be covered just once by $\mathcal{P}$. The body of the loop checks whether the $\mathcal{P}$ and $\mathcal{PCD}_F$ form a suitable composition. In this case, these sets are combined (line 11) to produce a new composition. The algorithm finishes once a solution is found.

As in POTI, user preferences define the order in which each combination of PCDs is produced. The Pareto order is used to ensure that the algorithm returns the next best solution.

6 Validating the approach

In order to validate our proposal we have implemented a prototype and conduct an experiment to:

1 measure the recovery time of compositions in the presence of faults

2 evaluate the compliance of the recovered composition with regard to the user preferences.

6.1 Experimental settings

We generate synthetic compositions and services to perform our experiment. This lets us explore a number of functionalities with different 'complexities' (in terms of number of services) and evaluate the scalability of our method. Algorithm 3 describes the workflow of the experiment. First, this algorithm defines the generation of synthetic cases for the experiment, and then it simulates failures and executes the recovery method.

Algorithm 3 evaluates compositions of different sizes, ranging from $F_{\min}$ to $F_{\max}$ functionalities. For each composition, every possible failure is simulated for N times. In our setting, we have explored compositions ranging from $F_{\min} = 4$ to $F_{\max} = 11$, being each failure simulated for $N = 5$ times.

For a given number of functionalities, Algorithm 3 generates the composition specification and a set of web services (lines 3 to 5). In the composition specification, the parameters x_j, x_{j+1} of each F_j represents, respectively, input and output of the functionality, so imposing a sequential behaviour to the composition. The registry of available services $\mathcal{W}$ is generated as the result of calling the function BuildSyntheticWebServices (Algorithm 4).

We next call the POTI algorithm, using the synthetic services to produce a set $\mathcal{C}$ of service compositions and the set $\mathcal{PCD}$ of PCDs generated along the rewriting process (line 6). These steps establish the basis for the simulation of failures and for the execution of our recovery method. Line 7 of Algorithm 3 calls the procedure SimulateFailures to run our healing algorithm.

Let us now look at the generation of synthetic web services, for a given number n of functionalities, as described by Algorithm 4. This algorithm takes a value representing the number of functionalities to be covered and returns a set $\mathcal{W}$ containing the specification of all services that can be defined to cover functionalities $F_1, \ldots, F_n$.

Algorithm 3 General recovery process

Input:
- The minimum number of functionalities F_{min}.
- The maximum number of functionalities F_{max}.
- The number of times N to repeat each recovery.

1: **procedure** Monitor(F_{min}, F_{max}, N)
2: **for each** $i \in \{F_{min}, \ldots, F_{max}\}$ **do**
3: $C_{spec} \leftarrow C(x_1, x_{i+1}) \equiv F_1(x_1, x_2), \ldots, F_i(x_i, x_{i+1})$
4: $\mathcal{W} \leftarrow$ BuildSyntheticWebServices(i)
5: $(\mathcal{C}, \mathcal{PCD}) \leftarrow$ POTI(C_{spec}, $\mathcal{W}$)
6: SimulateFailures($\mathcal{W}, \mathcal{C}, \mathcal{PCD}, N$)
7: **end for**
8: **end procedure**

Algorithm 4 begins by defining $\mathcal{W}$ as an empty registry, to be populated with the specifications of services $S_{[r,s]}$, where r and s are the first and last functionalities covered by the service. For instance, the service $S_{[2,4]}$ covers the interval of functionalities F_2 to F_4, being defined by the following specification:

$$S_{[2,4]}(x_2, x_5) \equiv F_2(x_2, x_3), F_3(x_3, x_4) \\ F_4(x_4, x_5)$$

In this algorithm, each iteration of the loop at line 3 controls the generation of web services for a given number i of functionalities. In this manner, the first iteration builds services with one functionality, the second iteration builds services with two functionalities, and so on. The loop at line 4 is in charge of generating all the services with a number i of functionalities, whereas the loop at line 6 specifies the functionalities for a given web service.

Let us now illustrate the construction of the synthetic cases of our experiment by assuming a number of four functionalities.

Example 8 (registry with services covering four functionalities): let us consider a composition with four functionalities, as described by the following specification:

$$C_{spec}(x_1, x_5) \equiv F_1(x_1, x_2), F_2(x_2, x_3), F_3(x_3, x_4), F_4(x_4, x_5)$$

where C_{spec} describes a composition with functionalities $F_1, \ldots, F_4$. For this specification, Algorithm 4 builds the synthetic services below:

- Services with 1 functionality:

$$S_{[1,1]}(x_1, x_2) \equiv F_1(x_1, x_2)$$

$$S_{[2,2]}(x_2, x_3) \equiv F_2(x_2, x_3)$$

$$S_{[3,3]}(x_3, x_4) \equiv F_3(x_3, x_4)$$

$S_{[4,4]}(x_4, x_5) \equiv F_4(x_4, x_5)$

- Services with 2 functionalities:

  $S_{[1,2]}(x_1, x_3) \equiv F_1(x_1, x_2), F_2(x_2, x_3)$

  $S_{[2,3]}(x_2, x_4) \equiv F_2(x_2, x_3), F_3(x_3, x_4)$

  $S_{[3,4]}(x_3, x_5) \equiv F_3(x_3, x_4), F_4(x_4, x_5)$

- Services with 3 functionalities:

  $S_{[1,3]}(x_1, x_4) \equiv F_1(x_1, x_2), F_2(x_2, x_3), F_3(x_3, x_4)$

  $S_{[2,4]}(x_2, x_5) \equiv F_2(x_2, x_3), F_3(x_3, x_4), F_4(x_4, x_5)$

- Services with 4 functionalities:

  $S_{[1,4]}(x_1, x_5) \equiv F_1(x_1, x_2), F_2(x_2, x_3), F_3(x_3, x_4), F_4(x_4, x_5)$

  Notice that these services cover all the combinations of functionalities F_1, …, F_4.

■

Given the specifications of a composition and of the available services, the rewriting algorithm POTI produces a set of service compositions $\mathcal{C}$ and a set $\mathcal{PCD}$, containing all the PCDs used to generate the compositions in $\mathcal{C}$.

Algorithm 4 Build synthetic web services

Input:

- The number of functionalities n.

Output:

- All the possible web services for n functionalities.

```
1: function BuildSyntheticWebServices(n)
2:     W ← ∅
3:     for each i ∈ {1, …, n} do
4:         for each j ∈ {1, …, n – i + 1} do
5:             F ← ∅
6:             for each k ∈ {j, …, j + i – 1} do
7:                 F ← F ∪ {F_k(x_k, x_{k+1})}
8:             end for
9:             W ← W ∪ {S_[j, j+i–1](x_j, x_{j+i}) ≡ F}
10:        end for
11:    end for
12:    return W
13: end function
```

Algorithm 5 is the core of our experiment. This algorithm defines the procedure SimulateFailures, called by Algorithm 3 (line 7). The procedure SimulateFailures takes:

1 the set (or registry) $\mathcal{W}$ of services

2 the set of compositions $\mathcal{C}$, generated by POTI

3 the set $\mathcal{PCD}$, also generated by POTI

4 the number N of times to execute each recovery.

In our case, we defined $N = 5$.

Algorithm 5 Simulation of failures

Input:

- The set $\mathcal{W}$ of available web services.
- The set $\mathcal{C}$ of service compositions.
- The set $\mathcal{PCD}$ of available PCDs.
- The number of times N to repeat each recovery.

1: **procedure** SimulateFailures($\mathcal{W}, \mathcal{C}, \mathcal{PCD}, N$)
2: **for each** $C \in \mathcal{C}$ such that
3: $C(x_1, x_{i+1}) \equiv$ $S_{[1,a]}(x_1, x_{a+1}),\ldots,S_{[b,l]}(x_b, x_{i+1})$
4: **do**
5: $\mathcal{PCD}_C \leftarrow$ PCDsOf(C)
6: **for each** $j \in \{1, \ldots, N\}$ **do**
7: SetPreferences ($\mathcal{PCD}, \mathcal{PCD}_C$)
8: **for each** $S_{[k,l]} \in \{S_{[1,a]},\ldots,S_{[b,i]}\}$ **do**
9: $\mathcal{L} \leftarrow \{S_{[a,b]} \in \mathcal{W} \mid k \le a, b \le l\}$
10: *Heal*($\mathcal{L}, \mathcal{PCD}_C, S_{[k,l]}$)
11: $\mathcal{P} \leftarrow \{S_{[a,b]} \in \mathcal{W} \mid k \le a \le l, b > l\}$
12: *Heal*($\mathcal{P}, \mathcal{PCD}_C, S_{[k,l]}$)
13: $\mathcal{T} \leftarrow \{S_{[a,b]} \in \mathcal{W} \mid a \ne k\}$
14: *Heal*($\mathcal{T}, \mathcal{PCD}_C, S_{[k,l]}$)
15: **end for**
16: **end for**
17: **end for**
18: end procedure

The procedure SimulateFailures tries to recover from failures in each composition C in $\mathcal{C}$ (line 2). Each composition C is assumed to be the composition with the highest preference during the simulation of their failures. In this way, the algorithm assigns the

maximum possible preference for C. The preference of a composition is calculated as the mean of the preference scores of its component services. Thus, we assign the value 1.0 to the services that participate in C, so its preference score will be 1.0. Then, Algorithm 5 randomly assigns preference scores between 0.01 and 0.99 to the remaining services (line 7). Thus, the remaining compositions will have a mean preference score smaller than 1.0.

The inner loop of Algorithm 5 simulates the failure of each service $S_{[k,l]}$ in the definition of C (line 8). The algorithm uses the available services for addressing the three levels of recovery, one after the other. This is done by filtering the set $\mathcal{W}$ to ensure that the right subset of services is offered to the procedure heal:

- the set $\mathcal{L}$ is a registry which contains all the services in $\mathcal{W}$ that may be used to substitute $S_{[k,l]}$ in $\mathcal{C}$ (i.e., to perform a local recovery of $S_{[k,l]}$)
- the set $\mathcal{P}$ is a registry containing all the services in $\mathcal{W}$ that may be used to substitute $S_{[k,l]}$ and all those services appearing in $\mathcal{C}$ after $S_{[k,l]}$] (i.e., to perform a partial recovery after the failure of $S_{[k,l]}$)
- the set $\mathcal{T}$ is a registry that contains all the services in $\mathcal{W}$ that may be used to build a new composition from scratch, not using $S_{[k,l]}$ (i.e., to perform a total recovery after the failure of $S_{[k,l]}$).

In our validation, the Heal procedure is feed with registry sets $\mathcal{L}, \mathcal{P}$ and $\mathcal{T}$. The time spent for each kind of recovery is registered.

Our experiments were executed on top of an Ubuntu 18.04 LTS Bionic Beaver, Linux kernel 4.15, 8GB RAM, AMD Phenom II X4 820 2.8GHz Quad-Core, Java 8. The size of the abstract compositions considered ranged from $F_{\min} = 4$ to $F_{\max} = 11$ distinct functionalities. The experiment was executed $N = 10$ times for each abstract composition.

6.2 Running the experiment

We investigate the time required for healing compositions at each recovery level, as well as the impact of recoveries on the preference value of compositions. The influence of the locality of the fault on the time cost and the preference impact is also investigated during the experiment. For each composition generated by POTI, we define its preference as 1.0 and then simulate failures at each of its component services. The recovery of each failure is tried at the local, partial and total levels, using, respectively, the registry sets $\mathcal{L}, \mathcal{P}$ and $\mathcal{F}$. After that, we calculate the preference value of the replacing composition. This strategy is applied ten times for each composition, in order to obtain a consistent time measurement. We experimented with compositions containing from 4 to 12 functionalities.

6.2.1 Recovery time

Figure 2 shows the average time taken by the healing algorithm to perform one local, partial or total recovery, for each size of composition. The recoveries executed in compositions of four functionalities are represented as more expensive than recoveries in some larger compositions. This behaviour is explained by the time spent by the Java virtual machine (JVM) to start up. As expected, local recoveries are less expensive than

partial and total recoveries. Indeed, the local recovery simply tries to substitute the failed service for other services providing the same functionalities.

Partial recoveries involve the substitution of the failed service, as well as the rebuilding of the part of the composition which has not been executed yet. Total recoveries imply in the substitution of the whole composition. Partial and total recoveries have a combinatorial nature, since they choose a combination of PCDs to cover more than one functionality. These facts explain the increase in the time of recoveries at these levels, as shown in Figure 2.

Figure 3 shows the average time spent for the execution of total recoveries for each composition size. As previously shown in Figure 2, the total recovery demands more time for recovering than the other levels of recovery. This difference is particularly evident in the cases where compositions with have ten or more functionalities. The data of Figure 3 considers the accumulation of time cost of the unsuccessful attempts of the previous levels of recovery, local and partial.

Figure 2 Average recovery time of all levels of recovery (see online version for colours)

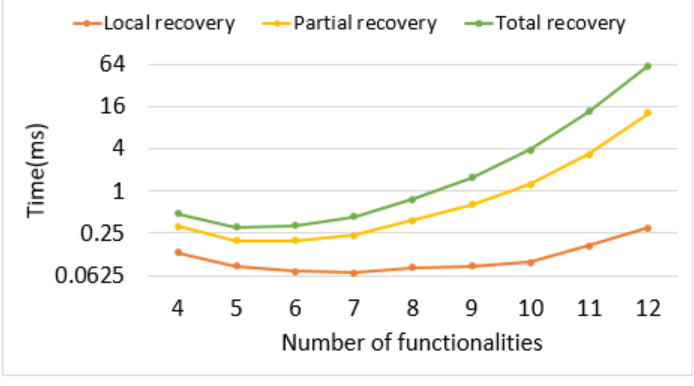


Figure 3 Total recovery – average recovery time (see online version for colours)

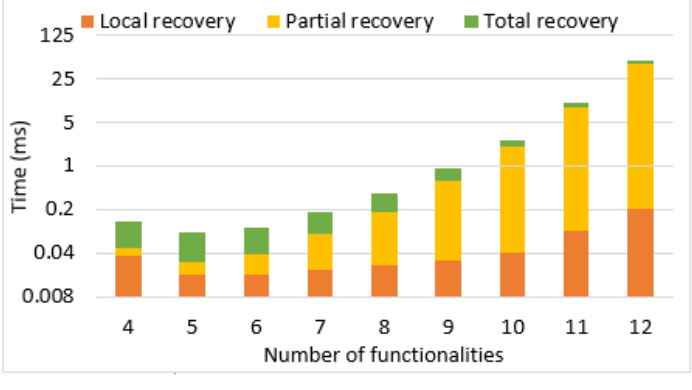


Figure 3 also shows that, for more complex compositions, the time spent to search a solution for total recovery is considerably smaller in comparison with the time taken by the algorithm to conclude the failed attempts of local and partial recovery. Indeed, the

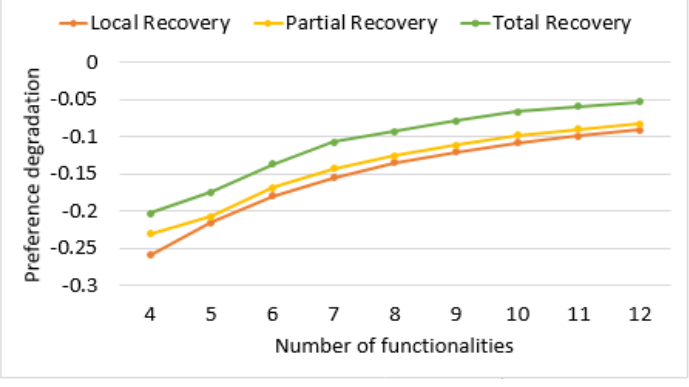
problem of recovering a service composition becomes more flexible when all services are eligible for substitution. This flexibility eases the search of solutions at the total level in comparison with the other levels that present restrictions for their combinatorial problems.

6.2.2 Preference degradation

This section explores the results in terms of user preference of the recovered compositions. Figure 4 shows the mean preference degradation achieve by the healing algorithm when performing local, partial or total recoveries, for each size of the composition. The values of degradation were obtained as the difference between 1.0 (the preference of each most preferred composition, as generated by POTI) and the mean preference value of the recovering composition produced for each level of recovery.

Note that, for all recovery levels, the degradation of preferences reduces as the number of functionalities increases. This behaviour is explained by the fact that the higher the number of functionalities, the smaller the relative contribution of the failed service for the overall preference value of a composition. We can also notice that local recovery consistently provokes a greater preference degradation. The partial recovery delivers better results but they are close to the ones reached by the local level. The best results are obtained by using the total recovery, which is more flexible since the algorithm may substitute all the services in the original composition.

Figure 4 Average preference degradation of all levels of recovery (see online version for colours)



Considering Figures 2 and 4 we observe that, for smaller compositions, the time cost does not differ significantly between the levels but the total recovery delivers the lowest preference degradation. Additionally, in the case of more complex compositions, the total recovery still deliver the smallest preference degradation. In all cases, the total recovery represents the most expensive level. As seen in Figure 3, the unsuccessful attempt of the partial recovery is the primary cost of the total recovery. These observations suggest that the adoption of total recoveries for small compositions is the best option.

In the case of larger compositions, the recovery mechanism could skip the attempt of partial recovery in order to reduce time costs. In that way, the local recovery is initially tried, but if this level of recovery is not successful the total recovery is initiated. These results also show that different levels of recovery may be suitable for different priorities on the time cost and preference degradation. The total recovery may be desirable for a

situation that prioritises the lower degradation of preference over the time cost. Whereas local recovery is suitable for cases that demand the fastest solutions for failures independently of the resulting preference degradation.

6.2.3 Locality of faults

This section is dedicated to analysing how the locality of the faults within the composition impacts on the recovery time and preference degradation.

In Figure 5, we show the average time cost for the total recovery depending on the locality of fault for different compositions. Notice that the time cost for total recovery reduces if the failed service is close to the end of the composition. Whereas faults in the initial portion of the composition demand more time for recovery.

Figure 5 Total recovery – average recovery time (ms) considering the locality of faults (see online version for colours)

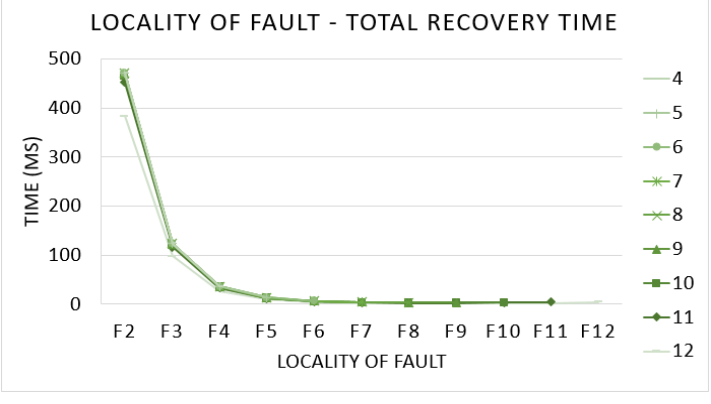
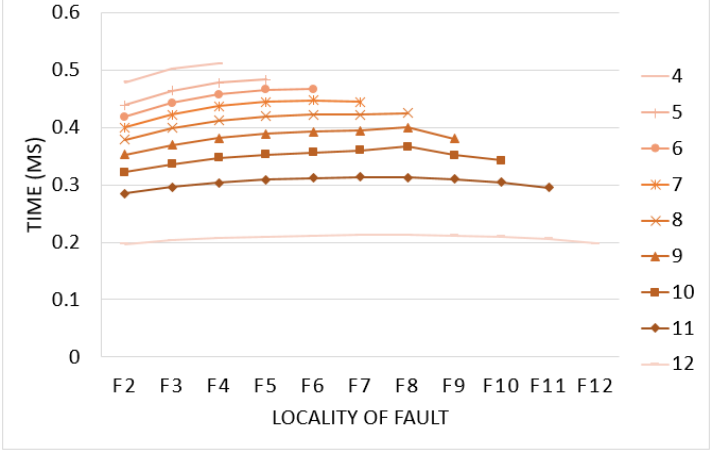


Figure 6 Total recovery – average recovery time (ms) considering the locality of faults – unsuccessful attempt of local recovery (see online version for colours)



We wanted to investigate the contribution of the attempts of other levels to the resulting time cost, considering the influence of the unsuccessful attempt of partial recovery on the final cost of total recovery. Figures 6, 7 and 8 show the time spent in each level of recovery while experimenting the total recovery of failures.

In Figure 6, we note that the same stable behaviour with regards to the locality of faults is maintained when reaching unsuccessful responses at the local level of recovery. These data also show that the local recovery has a minor influence on the average time cost of total recovery.

Figure 7 Total recovery – average recovery time (ms) considering the locality of faults – unsuccessful attempt of partial recovery (see online version for colours)

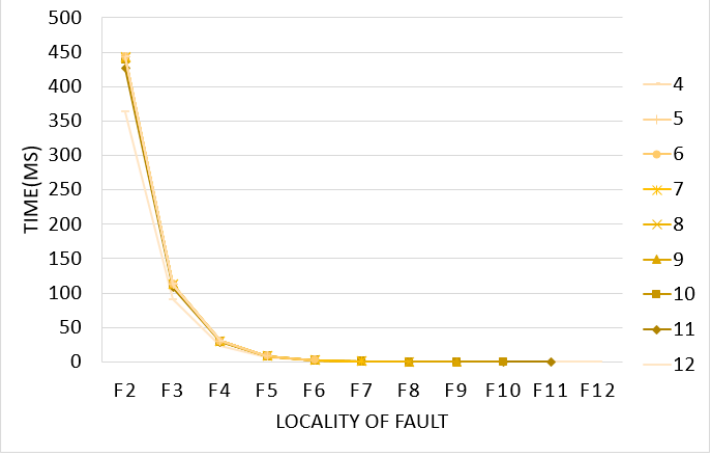


Figure 8 Total recovery – average recovery time (ms) considering the locality of faults – successful attempt of total recovery (see online version for colours)

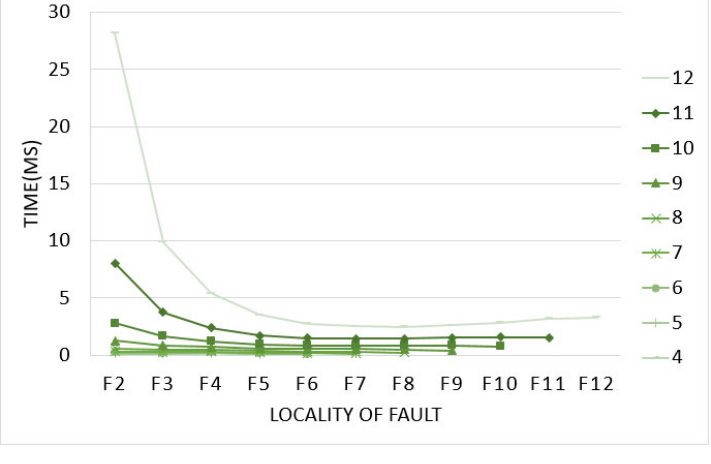


Figure 7 shows that failures that occurred at the beginning of the composition demands much more time for partial recovery. The values also enforce that the partial recovery represents the greatest contribution to the time cost of the overall time spent in total

recovery. Therefore, the partial level of recovery directly influences the time cost of total recovery with regards to the locality of faults. For example, for compositions with 12 functionalities, the time cost of a failure at the second service represents more than 360 ms than a failure at the last service of the composition.

Figure 8 shows the time spent to find a solution that may replace all the service composition on the total level. Note that for small compositions the time cost does not present a relevant difference when considering the failure in different portions of the composition. However, in the case of more complex compositions, when the first functionalities are involved in the failure, the time cost is slightly greater than the occurrence of failures at the last functionalities of the composition. For example, in the case of a composition of 12 functionalities, a failure at the second functionality would require 17 ms more than a failure at the last functionality.

7 Conclusions

In this paper, we have shown an overview of the research area of fault recovery of web service compositions. A taxonomy based on the locality of faults was presented as part of the research overview. We have proposed a recovery method based on the locality of the fault by considering different levels of impact caused by the substitution of components in the faulty composition. The proposed approach was validated by means of the execution of synthetic compositions and services, that explored a variety of scenarios of failures regarding the size the of initial composition, the user preference assigned to the involved services, and the locality of the fault occurred within the composition of services.

The recovery approach can be extended to address failures of services composition with other control flow patterns. We aim at extending our experimentation tools to develop a service-composition recovery testbed. The testbed will automatise the generation of synthetic compositions with different control flow patterns and an associated service registry. It will also provide a failure simulation environment that can follow the recovery process and estimate its cost within the execution.

References

Avizienis, A., Laprie, J-C. and Randell, B. (2001) 'Fundamental concepts of computer system dependability', in *Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, pp.1–16.

Ba, C., Cerqueira, T., Costa, U., Ferrari, M.H., Musicante, M.A. and Robert, S. (2016) 'Experiments on service composition refinement on the basis of preference-driven recommendation', *International Journal of Web and Grid Services*, Vol. 12, No. 2, pp.182–214.

Baresi, L., Guinea, S. and Pasquale, L. (2007) 'Self-healing BPEL processes with dynamo and the JBoss rule engine', in *International Workshop on Engineering of Software Services for Pervasive Environments: in Conjunction with the 6th ESEC/FSE Joint Meeting*, pp.11–20.

Barhamgi, M., Benslimane, D. and Medjahed, B. (2010) 'A query rewriting approach for web service composition', *IEEE Transactions on Services Computing*, Vol. 3, No. 3, pp.206–222.

Bianco, P., Lewis, G.A. and Merson, P. (2008) *Service Level Agreements in Service-Oriented Architecture Environments*, Technical Report, Carnegie-Mellon University, Software Engineering Inst., Pittsburgh PA.

Bruning, S., Weissleder, S. and Malek, M. (2007) 'A fault taxonomy for service-oriented architecture', in *10th IEEE High Assurance Systems Engineering Symposium, HASE'07*, IEEE, pp.367–368.

Chan, K.S.M., Bishop, J., Steyn, J., Baresi, L. and Guinea, S. (2007) 'A fault taxonomy for web service composition', in *International Conference on Service-Oriented Computing*, Springer, pp.363–375.

Costa, U.S., Ferrari, M.H., Musicante, M.A. and Robert, S. (2013) 'Automatic refinement of service compositions', in *International Conference on Web Engineering*, Springer, pp.400–407.

Erradi, A., Maheshwari, P. and Tosic, V. (2006) 'Recovery policies for enhancing web services reliability', in *International Conference on Web Services, ICWS'06*, IEEE, pp.189–196.

Fugini, M.G. and Mussi, E. (2006) 'Recovery of faulty web applications through service discovery', in *Proceedings of the 1st SMR-VLDB Workshop, Matchmaking and Approximate Semantic-Based Retrieval: Issues and Perspectives, 32nd International Conference on Very Large Databases*, pp.67–80.

Kießling, W. (2002) 'Foundations of preferences in database systems', in *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 20–23 August, pp.311–322.

Lemos, A.L., Daniel, F. and Benatallah, B. (2016) 'Web service composition: a survey of techniques and tools', *ACM Computing Surveys (CSUR)*, Vol. 48, No. 3, p.33.

Li, G., Liao, L., Song, D. and Zheng, Z. (2014) 'A fault-tolerant framework for QoS-aware web service composition via case-based reasoning', *International Journal of Web and Grid Services*, Vol. 10, No. 1, pp.80–99.

Liu, A., Li, Q., Huang, L. and Xiao, M. (2010) 'Facts: a framework for fault-tolerant composition of transactional web services', *IEEE Transactions on Services Computing*, Vol. 3, No. 1, pp.46–59.

Mesmoudi, A., Mrissa, M. and Hacid, M-S. (2011) 'Combining configuration and query rewriting for web service composition', in *2011 IEEE International Conference on Web Services (ICWS)*, IEEE, pp.113–120.

Orriëns, B., Yang, J. and Papazoglou, M. (2003) 'Model driven service composition', *Service-Oriented Computing-ICSOC 2003*, pp.75–90.

Papazoglou, M.P., Traverso, P., Dustdar, S. and Leymann, F. (2007) 'Service-oriented computing: state of the art and research challenges', *Computer*, Vol. 40, No. 11, pp.38–45.

Parra, D., Karatzoglou, A., Amatriain, X. and Yavuz, I. (2011) 'Implicit feedback recommendation via implicit-to-explicit ordinal logistic regression mapping', *Proceedings of the CARS-2011*, p.5.

Pottinger, R. and Halevy, A. (2001) 'Minicon: a scalable algorithm for answering queries using views', *The VLDB Journal – The International Journal on Very Large Data Bases*, Vol. 10, Nos. 2–3, pp.182–198.

Raines, G. (2009) *Cloud Computing and SOA*, MITRE, White Paper, October.

Shah, T.R. and Patel, S.V. (2016) 'A survey on issues and challenges of web service development, composition, discovery', *VNSGU Journal of Science and Technology*, July, Vol. 5, No. 1, pp.134–153, ISSN: 0975-5446.

Sheng, Q. et al. (2006) *Composite Web Services Provisioning in Dynamic Environments*, University of New South Wales.

Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S. and Xu, X. (2014) 'Web services composition: a decade's overview', *Information Sciences*, 1 October, Vol. 280, pp.218–238.

Simmonds, J., Ben-David, S. and Chechik, M. (2013) 'Monitoring and recovery for web service applications', *Computing*, Vol. 95, No. 3, pp.223–267.

Tian, G., Wang, Q., Wang, J., He, K., Zhao, W., Gao, P. and Peng, Y. (2019) 'Leveraging contextual information for cold-start web service recommendation', *Concurrency and Computation: Practice and Experience*.

Wang, G. and Yang, B. (2018) 'Failures handling strategies of web services composition base on Petri nets', in *International Conference on Intelligent Computing*, Springer, pp.608–617.

Wang, M.X., Bandara, K.Y. and Pahl, C. (2009) 'Integrated constraint violation handling for dynamic service composition', in *IEEE International Conference on Services Computing, SCC'09*, IEEE, pp.168–175.

Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M. and Russell, N. (2006) 'On the suitability of BPMN for business process modelling', in *International Conference on Business Process Management*, Springer, pp.161–176.

Yu, Q., Liu, X., Bouguettaya, A. and Medjahed, B. (2008) 'Deploying and managing web services: issues, solutions, and directions', *The VLDB Journal – The International Journal on Very Large Data Bases*, Vol. 17, No. 3, pp.537–572.

Yuan, Y., Zhang, W. and Zhang, X. (2018) 'A context-aware self-adaptation approach for web service composition', in *2018 3rd International Conference on Information Systems Engineering (ICISE)*, IEEE, pp.33–38.

Zhao, W.F., Liu, C.C. and Chen, J.L. (2012) 'Automatic composition of information-providing web services based on query rewriting', *Science China Information Sciences*, Vol. 55, No. 11, pp.2428–2444.