SOCA (2016) 10:413–435
DOI 10.1007/s11761-016-0193-6

ORIGINAL RESEARCH PAPER

FRWSC: a framework for robust Web service composition

Mohamed El Kholy[1] · **Ahmed El Fatatry**[1]

Received: 12 November 2015 / Revised: 7 January 2016 / Accepted: 19 March 2016 / Published online: 5 April 2016
© Springer-Verlag London 2016

Abstract The deployment of Web services in a highly dynamic environment brings about a number of research challenges. In dynamic Web services composition, failures and changes to atomic services cannot be detected before invocation. Hence, the failure or even the change in an atomic service may lead to the overall failure of the composite service. In addition, SOAP error code is not sufficient for the client to analyze the failure reason and handle it. In this work, we introduce a framework to deal with unexpected failures during runtime composition. The proposed framework is built on top of composite services stack as an interface between the composite service and its external service partners. The evaluation results show that by using the proposed framework, it is possible to avoid composite service failures that are caused by changes or failures in atomic services.

Keywords Web service composition · SOAP errors · Runtime failure · BPEL · Service replacement

1 Introduction

Web services composition is a promising technology that supports interoperability between heterogeneous business organizations. Web services are usually deployed in highly dynamic environments. In such environments, flexibility and reliability are key requirements. Web services composition aggregates different services from different organizations with various functional and nonfunctional properties. However, it is beyond the human ability to create the composition plan manually or to monitor its execution [1]. Artificial intelligence (AI)-based techniques have been extensively used in service composition research [2]. Ideally, given one or more goals and a set of available Web services, the task of an AI planner is to find a collection of Web services that can achieve the required goals [3,4]. This work focuses on building flexibility in composite services to overcome unexpected runtime failures. During the execution of the planned composite service, one or more services may fail to perform the required functionality various [5]. Such failures may lead to an overall failure of the composite service [6]. In many applications, especially in real-time application, it is not possible to stop execution to perform service maintenance. Recovery steps should be done on the fly.

In this work, we introduce a framework for robust Web service composition (FRWSC). The framework proposes a method to detect, analyze, and overcome individual service failure in composite Web services. To overcome a failure in such a dynamic environment, each service should be monitored. In a service-oriented environment, service execution takes place at the provider side. In such case, the consumer cannot monitor the actual service execution. Instead of monitoring the Web services execution, we propose monitoring its interface with the composite service. The interface between the composite service and its individual service is the SOAP messages [1,7,8]. In this work, we present a method to monitor the SOAP traffic to detect the behavior of each service and the possible failures [1,3]. Using this approach, it is possible to identify the failure reason and the possible solution. The composite service is considered as a closed system that interacts with the outside world through the interface of its services. The framework is built on top of the composite service as an extra interface between the composite service and

✉ Mohamed El Kholy
eng_mikholy@alexu.edu.eg

Ahmed El Fatatry
elfatatry@alexu.edu.eg

1 Institute of Graduate Studies and Research, Alexandria University, El Horia Road, Alexandria, Egypt

the external world. FRWSC is general and can be used for any composite service. However, we focus on the BPEL as the common language for orchestrating service composition [2–4]. In addition, we focus on SOAP as a common protocol to exchange messages in Web services [8]. The framework performs five tasks that facilitate building flexibility and reliability into composite Web services. The five tasks are listed as follows.

1. Detecting composite service failure.
2. Identifying failed processes.
3. Analyzing the type of failure and its reason(s).
4. Recovering failed services.
5. Resuming composite service execution.

The first step to fix a failure is to detect it. It is important to identify which process has failed and analyze both the type and the reason for the failure. Identifying the reason for the failure helps perform the correct solution to recover such failure [6]. Then, the execution of the composite service can be resumed safely.

This paper is organized as follows: Sect. 2 introduces the related work in the area of composite services failure and recovery. Section 3 specifies the problem of service failure and the solution requirements. Section 4 discusses failure types and solutions, and then maps each failure to one or more solutions. Section 5 discusses service replacement and dependency between services. Section 6 presents the structure of the proposed framework. The work is evaluated in Sect. 7 using a proof of concept prototype and experimental results. Finally, Sect. 8 discusses the conclusions and future work.

2 Related work

Approaches reported in the literature to deal with the problem of atomic service failure during composite service execution can be summarized into three main categories. Following is a discussion of the strengths and weaknesses of each approach.

2.1 First approach: software variability

Traditional software variability is based on two main concepts: variation points and variants. Variation points are the locations of expected failures. Variants are the alternatives of the expected errors. The aim of the variability approach is to combine traditional software variability with Web service composition technology [9]. Sun et al. in [9] propose a solution for the problem of service unavailability while executing a composite service by using variability management. Variation points are inserted at the invocation of each service. At each variant point, a set of alternative variant services that can be selected at such points are defined. At runtime, when a service becomes unavailable, a backup service with the same functionality can be invoked instead. Their work focuses on the design level and on introducing methods to insert variant points into software design. They introduced general extensions to UML diagrams to support modeling of variability using UML. Their work has been mapped to SOA using the extended UML diagrams. However, their work did not consider the dependencies between services. It also assumes that when two services have the same functionality, they can replace each other. Such assumption is not always correct. Two services may have the same functionality, but when composing them with other service, they may have a different behavior. This situation may occur because the two services require different messages as input, or require different preconditions. For instance, two services may perform the same functionality such as money transfer from a bank account. One service may require the client to perform a digital signature for his request while the other may not. Another limitation of the approach is that it does not define when to take the replacement decision. The work did not include the criteria for choosing the new service. Finally, the work does not support the case where there is no service that can perform the same functionality.

Software variability solution was discussed at the implementation level by Koning et al., in [10]. They introduced VxBPEL as an extension to BPEL language that allows variability management. VxBPEL contains additional XML elements that store variability information in a BPEL file. Variability information is defined inside the process definition. VxBPEL adds the variability information as extension elements using a different namespace. Their work added variation points to the BPEL code to support service replacement. They added new instructions (activities) to the BPEL file including variant point specification and then a set of variant services. These variant services can be invoked instead of the unavailable services. The main advantage of VxBPEL is that the variant services can be discovered dynamically. This allows dynamic changes to existing variation points in a VxBPEL process. However, the work is limited to the implementation level and on BPEL. Hence, the scope of the solution is narrow. Moreover, it does not analyze the reason of the failure and treats all failures in the same way. It does not provide a solution for the problem if the failure is associated with the client input parameters or precondition status.

2.2 Second approach: fault tolerance

This approach is more general and aims to deliver a correct service composition even in the presence of individual faults. The composed service is enriched with checkpoints that monitor the service execution. In case of a failure, all running processes terminate until the failed process is recov-

ered by substituting its service. Here we discuss two methods related to the fault-tolerance approach to solve the problem of individual service failure.

Liu et al. in [11] developed a framework to solve this challenging problem by allowing fault-tolerant composition of Web services. Their framework discussed how to deliver reliable service composition over individual unreliable services. They divide service failure into three main categories.

1. Logical failure is a type of failure that takes place when the business logic of the invoked service fails. For example, a service is required to book four tickets where there are only two tickets available.
2. System failure occurs when the server that hosts the service detects that the requested service does not exist.
3. Content failure occurs when the returned value of a previous service is not accurate. For example, wrong distance between two towns or wrong temperature degree.

Their approach to use fault tolerance to solve the service failure depends on monitoring the WSDL file and the BPEL file. Any specified operation is declared in a WSDL document [7]. Hence, the namespaces that are used in the WSDL document can be used as a unique name for failure. In this way, information about failure is collected from WSDL namespaces. Hence, the logical failure name is the name of the operation that failed in the WSDL file. At each operation, a checkpoint that has three possible values is created. These values are (completed, failed, canceled). According to the check value, a decision is taken. BPEL file is used to detect system failure. When a failure is detected, the composite service stops and a service replacement procedure is run. The work depends on the BPEL exception handling method to detect content failure. Such method is not effective because business rules undergo continuous changes. Hence, the throw–catch expressions have to be changed every time business rules change.

Omid Bushehrian et al. in [6] propose that fault tolerance for composite services should be handled at a higher level of abstraction. They built a language-independent method for Web service fault tolerance at the workflow level. In case of failure, the workflow applies a rollback operation. The rollback operation removes any effect of already executed service that has dependency relationship with the failed service. It also locks any running service and rolls back all services to the last dependent point. The work presents a good contribution to remove the effect of the executed service in case of failure. However, it does not provide a complete solution for the problem of service failure. It did not provide details about replacement methods or how to resume execution. It also did not deal with the local variables and how to store them. They apply a rollback mechanism to reach the last executed dependent service. Hence, after the composite service resumes execution, it must execute all the rolled back services.

2.3 The third approach: service replication

The replication approach has been used in solving failure problems associated with distributed systems. According to [12], only three countries provide 55.5 % of the available Web services. This means that the countries that consume these services will be affected by any Internet disconnections to these three countries. One solution to solve this problem is to have a set of replicate services in different physical places all over the world. There are two main types of replication: active replication and passive replication. Active replication means that the user requests are processed by all service replicas. Service requests should be processed in the same order in all the replicas to guarantee that all replicas execute the service request while having the same state. Hence, all replicas will produce the same output. If one replica failed under any condition, the rest of the replicas will provide the same functionality in a transparent manner to the requester.

The other type is passive replication which means that one replica service acts as master and another acts as backup. Hence, only the primary service will execute the client request. Upon the failure of the primary service, one of the backup services will take its place. Zibin Zheng and Michael R. Lyu [12] built a hybrid strategy of services replication. Their strategy combines both time redundancy and space redundancy. They divided the space redundancy into passive and active redundancy. The combination between time redundancy, passive redundancy, and active redundancy created nine possible solutions. Then they analyzed the conditions of utilization of each combined solution.

Jorge Salas et al. [13] point out that the availability of Web services must be guaranteed in case of failures and network disconnections. They created a Web service replication framework that solves the problem of service failure by replicating Web services. Their framework allows active replication of Web services in a transparent way to the client. The client sends a request to WS-Replication framework which internally transforms the request into several requests to the service replicas. Their framework contains a multicast component that provides group communication based on SOAP. A controller monitors these SOAP messages to make sure that the right result is sent to the client. Replication approach can help solve the service failure due to bad network connections or service inexistence at the suggested location (service access failure) [14]. However, it cannot solve the problem of service failure due to functional, semantic, security, data matching, and logical failure issues. Moreover, it is very complex and increases the service cost to have multiple copies of each service.

3 Problem statement and solution requirements

3.1 Problem statement

The first problem discussed in this work is the failure of a composite service as a result of change in one or more of its individual services [6,9,14]. Web services composition has proved to be difficult to maintain when one of its individual services fails or changes its interface [6,9,14]. After planning the composition, the specified services are discovered and then selected according to the defined criteria [15]. At runtime, binding takes place between the selected service and its concrete functionality. In the dynamic environment of the Web services, some services may fail to perform their functionality. Other services may change their interface. As the Web services are invoked on the fly, failures and changes are difficult to detect before invocation. Consequently, the failure or even the change in an individual service may lead to the overall failure of the composite service.

The second problem is that the SOAP error code is not sufficient for the client to analyze the failure reason and handle it [16]. Also, SOAP messages cannot detect semantic failures in the client's request or the provider's reply. For instance, SOAP exception handler cannot detect unrealistic temperature degree as shown in the following example. A SOAP request may have no response at all. To clarify this problem, we implemented a Web service that has two operations. The first operation returns the temperature of capital cities of Mediterranean countries. The service takes the short name of the country as input, and then returns the temperature of its capital. This service was invoked with wrong input parameters and the response of each invocation was monitored. Table 1 shows a number of wrong parameters and their respective responses. It is clear that that although the input parameter is wrong, no explicit error is returned back. The SOAP response contains implicit errors that cannot be detected by the user. The second operation returns the sum of two integers. Invoking the service with wrong parameters, the SOAP error response is the same. Hence, in a composite service, the user cannot clearly specify the invocation error to repair it. Table 2 presents the SOAP response for wrong invocations.

Table 1 Implicit SOAP error codes for different wrong input parameters

Input parameter	SOAP response
No input at all	No error, temperature=0
Wrong string	No error, temperature=0
Integer	No error, temperature=0

Table 2 Explicit SOAP errors for different wrong input parameters

Input parameter	SOAP response
First input is digit and the second is null	HTTP error 505
First input is digit and the second is float	HTTP error 505
First input is digit and the second is string	HTTP error 505
First input is Null and the second is digit	HTTP error 505

3.2 Solution requirement

RQ1

There is a need for a mechanism to monitor the outgoing SOAP messages of a composite service and validate it. The parameters included in the SOAP request should be compatible with the required inputs of the invoked service. Any wrong parameters such as type mismatch or different number of parameters will lead to service failure. For instance, a service that requires two integer values as input should not be invoked with only one integer. Also, it cannot be invoked with two string values.

RQ2

There is a need for monitoring the incoming SOAP response and differentiating between failed and successful invocations. SOAP error may be an explicit error or implicit error. Explicit error should be declared in the form of failure message informing the client that his invocation has failed. The implicit error may not be declared and is in the form of traditional SOAP reply but including a wrong result. This has to be treated.

RQ3

There is a need for a mechanism to specify and analyze the failure reason. The failures which are detected through monitoring the SOAP traffic should be analyzed to determine the type of each failure and its reason.

RQ4

There is a need for a mechanism to create a recovery plan to maintain failures in composite services. Such plan should consider the dependency between services and minimize the repair time.

RQ5

The composite service should safely resume execution after the recovery of individual service failure.

3.3 Motivation example

This section presents an example of a composite service failure as a result of failure or change in one of its atomic services. The "Travel planner" composite service aggregates different services from different organizations to arrange

trips. Each atomic service in the "Travel planner" is independent from other services. However, when these services are aggregated together, composition dependencies issues appear. For instance, the output of one service could be the input to another service. Any failure or change in one of the atomic services will lead to an overall failure of the composite service. In addition, the failure may not be reported to the client to avoid its reason in the next invocation. The failure reason could be wrong parameters that are provided by the user, for instance wrong name of the destination town. A more complex reason: one of the component services changes its input or output parameters. Such situation will affect the previous or successive service of the changed one.

4 Failure reasons

In this section, the different reasons for failures of individual Web services are analyzed. The analysis helps in developing methods to deal with the impact of individual service failure on the composite service. The individual service failure recovery is more efficient than replacing it because selection and discovery phases will not be repeated. Moreover, service replacement may lead to composite service failure. Such situation takes place if the new service is not adapted to the successive services in the composite service workflow. The new service is considered to be adapted to the successive service if it satisfies the following demands. First, the output parameters of the new service should be the same as the input parameters of its successive service. Second, the effect of the new service should satisfy the precondition of the successive services. In order to overcome a Web service failure, the reasons and location of the failure should be clearly specified. In this section, the process of service invocation is traced from its beginning to end.

The proposed framework divides the service failure into three main categories according to the location of failure. Then the reasons of failure at each location are analyzed by tracing all processes that occur at each location. The three main categories are client-side errors, communication errors, and provider-side errors. The first failure location can be at the client side where the service invocation starts. The client creates a service request that encapsulates his input parameters in a SOAP message. The second failure location can be at the communication path between the client and the provider. Such category includes the transformation of client request over the Internet to the service provider side and the transformation of the results back to the provider. The third location is the provider side where client parameters are extracted from the request message. Then the service execution takes place and the result is sent back to the client side

4.1 Client request

Web service invocation process starts with a client request in the form of SOAP message [8]. The client request encapsulates service input parameters that are extracted from the WSDL file [1,16]. Such process is complex to be done manually. Hence, an automated client-side interface is responsible for extracting service requirements and generating the request message. The client requests are sent to service providers in the form of SOAP messages over HTTP protocol. In case of invocation success, the client-side interface receives the SOAP reply from the service provider. The result parameters are extracted and then passed to the client. SOAP protocol is limited to exchange message between two end points (requester and provider) [17]. The exact data type specification for invocation parameters is offered by WSDL file. WSDL supports any type definition [1,7,16]; most services use XML Schema Definition (XSD) to define service parameters such as (int, string) [7]. The WSDL file can also import external definition using "import" element that specifies the location of external schema definition. Invocation problems occur when a service changes its required input parameters. For instance, a parameter with type float is changed into int type. Moreover, the imported types in WSDL file may be changed at its external locations without being locally changed at the WSDL file. The client-side interface uses the parameters extracted from the WSDL file to invoke the service [17]. The client-side interface is not aware of the change in service parameters or with external definition changes. Hence, service invocation is done using old parameters which are not compatible with the new service requirements. In such cases, a service failure may take place due to clients' requests. The SOAP errors that the client receives do not clearly describe the failure reason as mentioned in Sect. 3.1. Such problem worsens by increasing the number of invoked services. The client request may contain one or more of the following errors.

- Wrong number of input parameters.
- Wrong type of input parameters.
- Time delay for sending input parameters in synchronous service invocation.

4.2 Communication between service client and service provider

Web services technology relies on Internet connections and protocols to exchange messages between the consumer and the provider. Hence, it may suffer from connection problems at different network layers [4,12]. Although this paper focuses mainly at the application layer, this section discusses the connection failures at the other layers to present

Table 3 Data size in request and reply message for the first service

	Size of request message	Size of reply message
Empty SOAP message “with no header and no body element”	117 bytes	325 bytes
SOAP message containing null value of client parameters	279 bytes	1664 bytes
SOAP message containing 2 bytes (client request)	281 bytes	325 bytes
SOAP message containing 4 bytes (client request)	283 bytes	326 bytes

Table 4 Data size in request and reply message for the second service

	Size of request message	Size of reply message
Empty SOAP message “with no header and no body element’	117 bytes	325 bytes
SOAP message containing null value of client parameters	2134 bytes	1520 bytes
SOAP message containing 25 bytes of client request	2134 bytes	943 bytes
SOAP message containing 50 bytes of client request	2159 bytes	968 bytes

its impact on service failure. Web services is considered as a remote procedure call (RPC) between heterogonous systems [8]. However, Web services require additional features over traditional RPC to be able to communicate in platform-independent environment. The request message sent from the client should be able to interact with any environment. Hence, there is a need for a platform-independent protocol to transfer data. SOAP protocol has risen as the common protocol to exchange Web services messages [16]. SOAP offers the flexibility features needed to support the communication of heterogeneous systems [8,16]. SOAP does not have any restrictions about the data types included in the message body and can use any XML schema definition [17]. Also, transferring the client request by SOAP messaging does not limit the client to any rate for requests. The client can perform any number of requests per second according to the available speed. Other messaging protocols such as REST limits the client to 15 requests per second [16]. SOAP messages can be transferred over any transport protocol such as HTTP, SMTP, and MQSeries [16,17]. HTTP is the most widely used protocol to transfer SOAP messages [16]. The main advantage of using HTTP is that the SOAP envelope is integrated into a standard HTTP request. Hence, the service interaction is done through port 80 and is interpreted as a Web request by all firewall systems. Hence, using SOAP messages over HTTP does not need any more reconfigurations for communication management and security policies [17]. Moreover, HTTP is a request response-based protocol, so it is suitable to be used with Web services which belong to the RPC paradigm. Despite these benefits, SOAP over HTTP is the suitable protocol to exchange messages in Web services paradigm [16].

The simplicity and flexibility features of SOAP protocol have a number of drawbacks. SOAP has poor performance and is loaded with overhead even in case of transferring small sizes of data [8]. To analyze the performance of SOAP protocol, two Web services were built with different sizes of parameter data. A client-side interface was built for each service to send and receive data. The size of data transferred between the sender and the provider was calculated. The first service requires two input parameters of type integer and returns one output of the same type. The second service requires 50 inputs of type double and returns 25 double numbers. Then, the two services were invoked with different sizes of input data. The SOAP messages were then captured to calculate its size. Tables 3, 4 shows the overheads added to client data to be sent as SOAP messages. The overhead added to SOAP messages to be delivered by HTTP protocol is given in Table 2. Table 3 shows that to send only 4 bytes of client data the message size has to be increased to 281 bytes. Such problem is scalable when the size of client data increases. Table 4 shows that to send 50 bytes of client data the message size increases to 2.1 K bytes.

The delay time in SOAP messaging has many reasons. The SOAP is transferred over HTTP which requires establishing TCP connection before starting to send its packets [8]. Thus, a hand check method has to take place between the client and the provider before sending any data [16]. In addition, the client must wait after sending packets until receiving acknowledgment message from the provider. Then, after the client finishes sending all the required requests it has to wait until the provider ends the connection session [16]. The provider waits for the maximum segment life time before it closes the session. Hence, the client cannot start any new connection or send any new requests before receiving end of session message.

The impact of SOAP delay is more evident in wireless connections. SOAP over HTTP suffers from problems in the TCP protocol features which result in decreasing the sending rate in the case of segment losses [17]. Hence, frame loss due

to wireless connections will increase the SOAP delay [16]. SOAP delay leads to SOAP failure when the delay increases the threshold time of waiting for the reply message. In other cases, the delay leads to incorrect SOAP order which leads to implicit service error [8]. Thus the message transfer suffers from failure at SOAP level, HTTP level, TCP level, and data link level. The error produced from failed invocation is either SOAP error or HTTP error. Since the SOAP protocol is carried over HTTP, any communication error at the underlying network layers will be recognized as HTTP error [17], while SOAP errors imply that the data were transferred safely until HTTP layer, and the failure occurred at the client side or the provider side.

4.3 Provider-side error

When the client request arrives at the provider side, the provider starts to extract the client parameters which are encapsulated in the SOAP message [17]. The XML SOAP message is parsed and interpreted. There are many different types of XML parsers that the provider can use [16]. The schema definition XSD that is included in the SOAP message provides the needed basic types for encoding different data types of the client parameters [8]. Hence, the provider maps the client parameters to the corresponding input in the provider code. Then, the client executes its service, and the result value is then encapsulated into a SOAP message to be sent back to the client. Failures that may take place at the provider side can be summarized as follows.

1. Unavailable provider.
2. The XML parser that is used with the provider is incompatible with the client SOAP message.
3. Errors in the provider code that lead to service failure.
4. Errors in encapsulating the result in a SOAP message (such as XSD errors).
5. Errors in binding SOAP reply to HTTP protocol.

5 Service replacement

Composite Web services are executed in an open, unreliable, and rapidly changing environment [18]. Hence, individual services may be temporarily unavailable due to network problems. Other services may be deleted from service registries as their providers stop offering them [19]. Moreover, many services may change their interface, requirements, or QoS parameters [18]. In such cases, replacement of individual services takes place. Service replacement may also be done to substitute a service that fails to fulfill the required functionality. Hence, service replacement is a continuous activity in the composite service life cycle. Composite Web services are commonly composed of a large number of services [20,21]. Such services are developed by different teams with their own goals. Each service encapsulates its functional description and interface requirements. Service composition results in dependencies between the constituent services. A number of approaches for solving the issue of service dependency have been reported in the literature. These approaches focus on the data dependency extracted from input and output parameters [21]. Most of these approaches assume that the data types are the same in all service descriptions [22]. In this work, we differentiate between different data types according to the schema definition used in each XML file that describes each service. In addition, this work considers different dependency types such as control and precondition dependency.

There are two solutions to avoid failure while replacing an individual service with another one. The first solution is to choose a service identical to the replaced one. We define identical services as follows: two services are identical if they have the same functionality, the same parameters (inputs and outputs), the same preconditions, and the same postconditions. The two services should also be described by using the same schema definition (XSD).

The second solution is to adapt the new service to the execution context of the composite service. In such case, different dependencies between services should be detected to ensure safe resuming after service substitution. The issue of dependency in Web services differs from dependency in traditional software because the source code of the service is not available in case of Web services [23,24]. This work addresses the problems that are caused as a result of having interdepended services by parsing WSDL files to create UML diagrams. Then these UML diagrams (class diagrams) are used to detect dependencies between services [25]. The reason for choosing UML is that it is the standard modeling language for OO systems. This approach can benefit from analytical studies that are concerned with dependencies in UML diagrams. Another benefit from transforming WSDL file to UML diagrams is that the UML diagrams are human readable and understandable [25], rather than WSDL files which are not suitable for human use. To achieve this, there should be mapping mechanism between services and classes in UML diagrams.

5.1 Types of dependencies between services

Discovering dependency between Web services helps in two points. The first is to detect the impact of a service failure on other services. The second helps to understand the reason of service failure by tracing its dependent services. In this work, the service dependencies are divided into two types: data dependency and precondition dependency.

5.1.1 Data dependency

The input and output parameter types are extracted from the WSDL file of each service. In order to exchange parameters between services, the parameter types should be compatible. The key challenge in Web service parameter description is that there are many type definitions. In traditional OOP, each language has its own types [25]. However, in Web service environments, services parameters (input, output types) are described by different schemas according to the used XSD [26]. In order to pass the output parameters from one service to be used as input for another service, the transferred parameters should be compatible [27]. The number of outputs should be the same as the inputs of the successive service, as well as the parameter types and their definitions [27,28]. XSD includes attribute declaration which define the name and the type of each attribute. XSD provides several different primitive data types that constrain the textual value of each attribute [28]. XML provides the flexibility to constrain the values of each attribute [28]. A case in point is using a valid date, or a specific range of numbers. Such flexibility features is included in the XSD of the XML file. The XSD provides the facility to construct complex types from the primitive data types. Hence, each service description that is written in XML file has its own type definition according to the used XSD.

5.1.2 Precondition dependency

Each Web service has its own requirements to offer its functionality [29]. It first needs to be invoked by an invocation message from a trusted node. Invocation should be done according to the service interface description [27,29]. The interface description includes the required input parameters. These requirements are considered as explicit requirements. Behind this, there are a set of preconditions that should be satisfied before service invocation takes place. In case of invoking a single service, it may be easy to detect these preconditions before invocation, or to detect it from the error messages. By considering a composite service that aggregates a large number of services, precondition should be considered in both the design and the implementation phases [30]. In case of service replacement after composition, precondition parameters may lead to failure in two different ways: first, if the new service requires precondition that differs from the replaced service; second, when the new service has implicit effects that differ from the replaced service and these effects are needed for the successive service.

5.2 Extracting UML diagrams from WSDL files

UML diagrams can be used to formally describe Web service composition [25]. UML-S has been introduced as an extension to UML to formally describe Web services. In the literature, many researchers suggest using UML-S class diagram to model Web services [25]. The UML-S class diagrams model the service as a set of methods that represent the service operations [31]. Then the class attributes represent the service parameters. Such modeling is suitable to describe individual services in a static mode [25]. The proposed work reverses the modeling process by extracting the UML class diagram from the service description. Such process is done by parsing the WSDL file of each service to build a supporting class diagram. Class diagrams are constructed to allow solving dependency problems using both manual and automated methods [31]. (Manual methods can be used such as UML diagrams which are human readable). Automated methods can use dependency solving programs that take UML diagrams as inputs. UML-S class diagrams provide useful information that can help in solving dependency problems in static mode [31]. However, class diagram cannot support dynamic interaction between services in a composite service. To help modeling of dynamic interaction between service activity, the proposed work extends the use of UML-S to use activity diagrams.

The WSDL file describes Web services in terms of ports that include one or more operations [7]. Parsing the WSDL file reveals the tag “interface name” which includes the service name [1]. Then, the tag “operation name” that indicates the methods of the class representing the service [1]. The service parameters are extracted from the “types” tag that includes both simple types and complex types. These parameters are mapped to the attributes of the service class diagram. The class diagrams model the service interface and service parameters in a static mode [7]. Figure 1 shows the transformation from WSDL file to service class diagram. The interaction between the service and the control flow of the composite service are modeled by activity diagram. The activity diagrams are extracted from the composition language (BPEL) [23]. BPEL includes “basic activities” such as wait, invoke, and reply [24]. The control flow is done by “structure activities” such as flow, sequence, assign and split. Parsing a BPEL file maps each invocation process to an activity [23], while the control flow will be mapped to each corresponding meaning in the activity diagram. For example, sequence activity in BPEL is mapped to serial execution while flow is mapped to parallel execution.

6 Structure of FRWSC

The proposed framework is built on top of composite services stack as an interface between the composite service and its external service partners. FRWSC is a lightweight general framework that can be implemented in any composition language. The proposed framework monitors SOAP

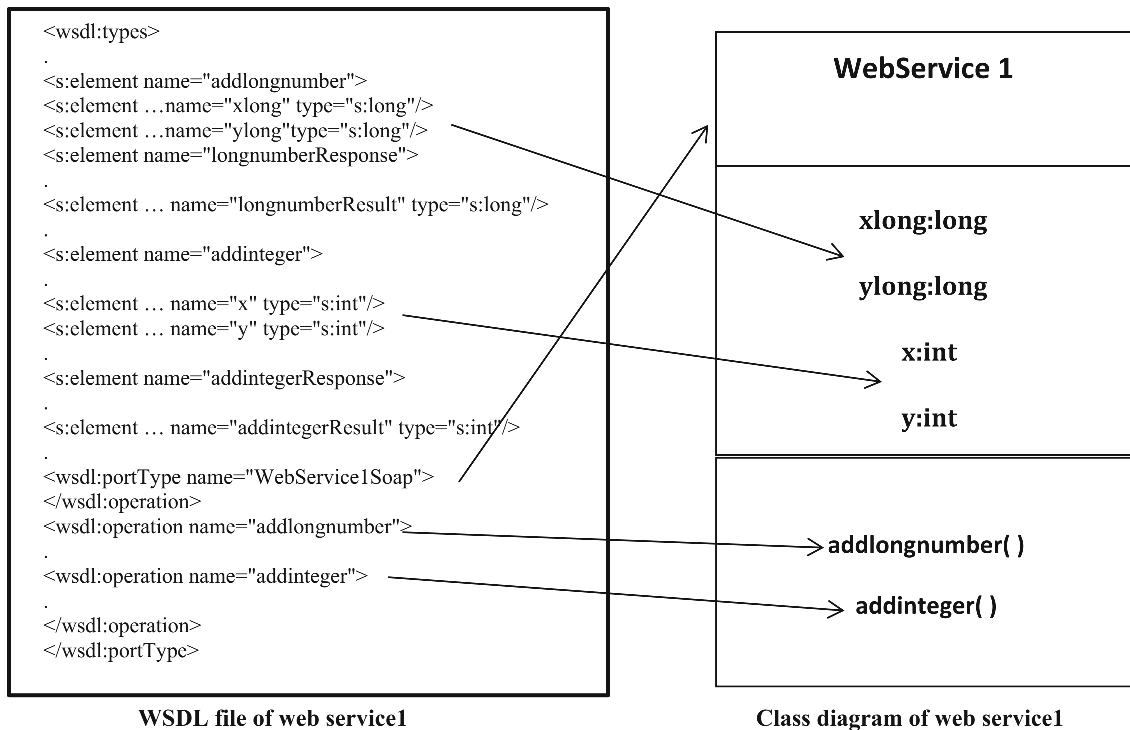

Fig. 1 Transformation from WSDL file to class diagram

traffic to detect any service failure. The detected failure is analyzed to specify the failure reason and its type. The reason is then passed to a solution provider to find the suitable solution. FRWSC consists of two main parts that interact together to detect SOAP failures and overcome these failures. The first part is the failure detector, while the second is the solution provider. The failure detector monitors SOAP messages and detects both explicit and implicit errors. The explicit errors include SOAP error messages and time out errors. The implicit errors include the errors in the content of SOAP messages without any error messages. For instance, a temperature service that returns a value over 80C degrees is considered an implicit error. Explicit errors are the errors which are supported with error messages such as "server busy." Figure 2 shows the sequence of the failure detector and its interaction with SOAP messages. The second part of our framework is the solution provider. The solution provider receives the SOAP failure and its type from the failure detector. The solution provider uses AI methods to analyze the failures to find the optimal solution.

6.1 Failure detector

The failure detector monitors the execution of the running composite service. Every captured SOAP message is used to detect unexpected failures of composite service activities.

Step one

Before the failure detectors starts, its database is enriched with all the composite service activities to be monitored. The database is also enriched with all the known SOAP errors as well as all the ranges of the input and output contents of SOAP messages. Such data are extracted from the WSDL file of each service.

Step two

The SOAP traffic is captured and the failure detector checks the parameters of the invoked service that are sent in a SOAP request. If the parameters are out of range of the values in the requirements of the invoked service, then a service failure is predicted. Hence, it is possible to predict the service failure before invocation. In this case, the execution is terminated and the failure reason is passed to the solution provider. Otherwise, the service invocation is performed.

Step three

The failure detector calculates the time before the reply message is sent back. If the waiting time exceeds a threshold time, the composite service is terminated. Then, the invoked service is passed to the solution provider with the reason of failure. If the time violates the response time included in the QoS properties, a violation report is produced, and the composite service completes execution.

Step four

The SOAP message of the service reply is checked for known errors. If the reply is an error message, the execu-

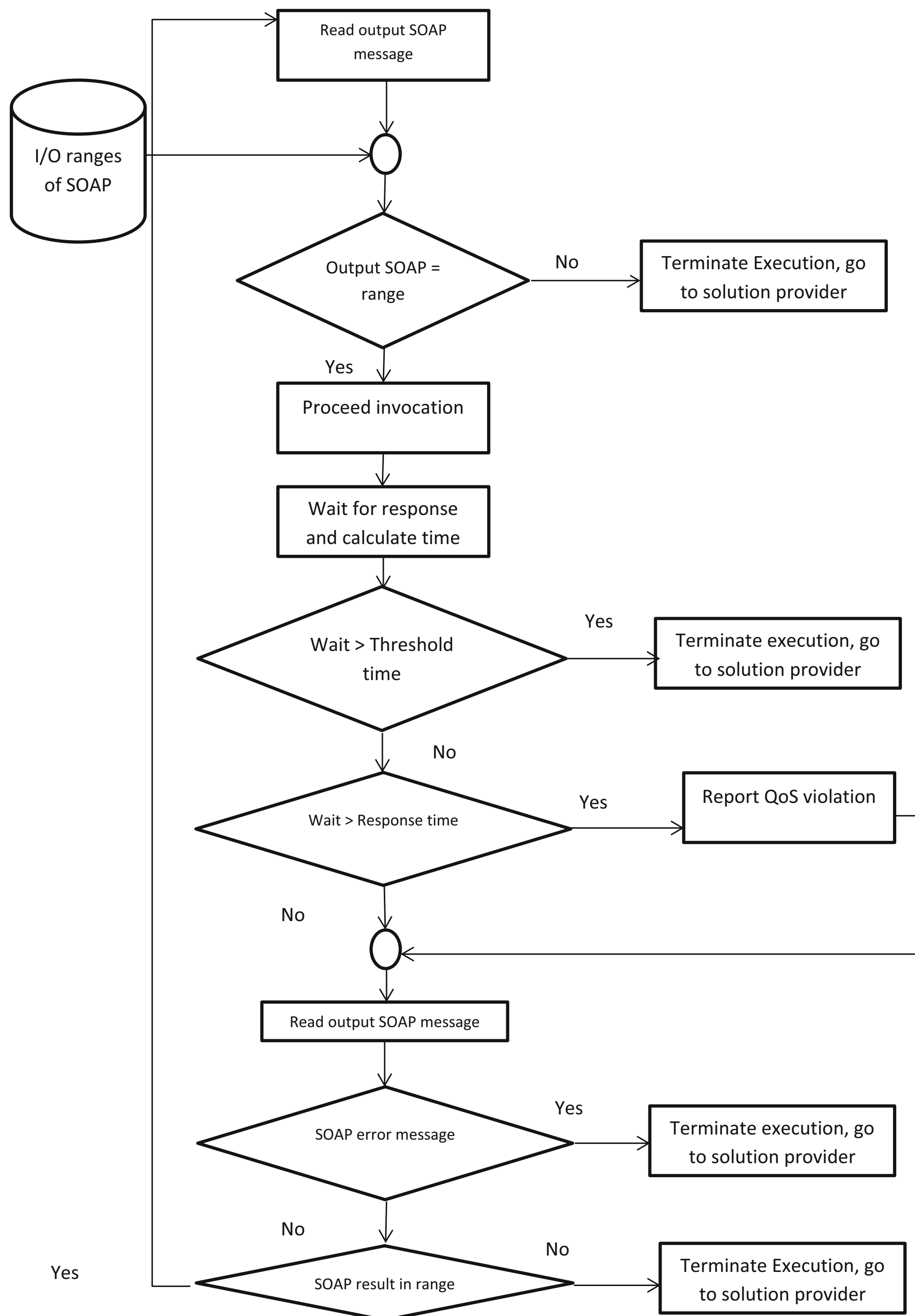

Fig. 2 Detecting service failure by monitoring SOAP Message

tion is terminated, and the service and its failure reasons are passed to the solution provider. Otherwise, the content parameters are checked, if they are in the expected range, the service execution is completed and the next service begins execution. If the reply parameters are out of range, the execution terminates and the service and its failure reason are then passed to the solution provider.

6.2 Solution provider

The aim of the solution provider is to find the optimal solution for unexpected failures that occur at runtime. The open environment in which the composite services run brings about different restrictions on the provided solution. Different services from different organizations are aggregated together.

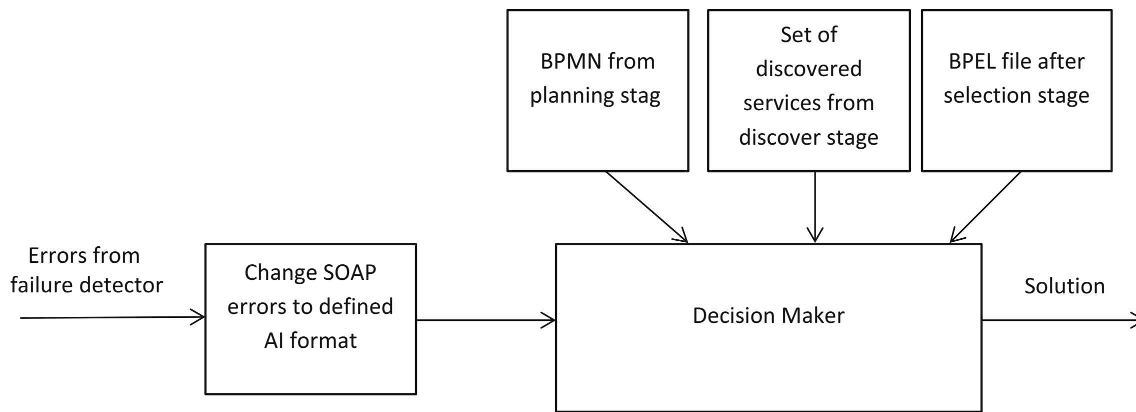

Fig. 3 Structure of the solution provider

The change in any individual service has an impact on the others services and on the overall execution of the composite service. The failure solutions are divided into three main categories according to their impact on other services in the composite service.

6.2.1 Solution types

- Simple solution

The simple solution has a limited effect on other services as it does not perform service substitution. Such effect is limited to the delay of service execution that may affect another service waiting for output parameters. The simple solution includes the following three actions.

1. Invoking the same service after correcting the out coming SOAP message.
2. Invoking the same service after a period of time.
3. Invoking the same service from another place.

- Intermediate solution

This solution includes the replacement of the service with another equivalent service. Even if the equivalent service has the same functionality as the replaced one, the two services may differ in parameters or precondition requirements or in QoS. Such changes may affect the other services which interact with the service that has been replaced. In this solution, a list of all interacting services with the replaced service is created. Then, the data exchange between these services is analyzed to determine the impact of change. In such a case, the change impact is examined at the implementation level.

- Complex solution

This solution is last choice to overcome service failure. The composite service is rolled back to the planning stage to specify a new process. Then, a new process is mapped to a new service. The new service is inserted inside the composite service to replace the failed service. In this case, the impact of adding the new service is analyzed at the design level which leads to more complex recovery.

6.2.2 Structure of the solution provider

The solution provider is supported with an AI machine that helps in the decision-making process. After composing the composite service, the AI machine is enriched with the Business Process Modeling Notation (BPMN), which represents the modeling notations of the composite service, as well as the BPEL file. Figure 3 shows the structure of the solution provider and its inputs and outputs. At runtime, the solution provider accepts the names of the failed services and the reason of each failure. The SOAP errors are analyzed with the AI programs to produce the optimal solution. The decision maker solves the failure starting with the simplest solution. It begins with simple solution then intermediate solution and at last: the complex solution. For each solution, the impact analysis is performed to ensure safe recovery before resuming execution (Fig. 4).

7 Empirical evaluation of the proposed framework

The research in the area of Web service composition can be evaluated from different perspectives such as performance, security, and cost [24]. However, in this work, the perspective of the evaluation is the flexibility of composite services. The proposed framework addresses the problem of failures in individual services during dynamic service composition. To evaluate such work, we analyzed the behavior of composite services against different failure situations. This section evaluates the response of the composite service to changes or failures in its atomic services. We created different situations of atomic service failures and measured the possibility

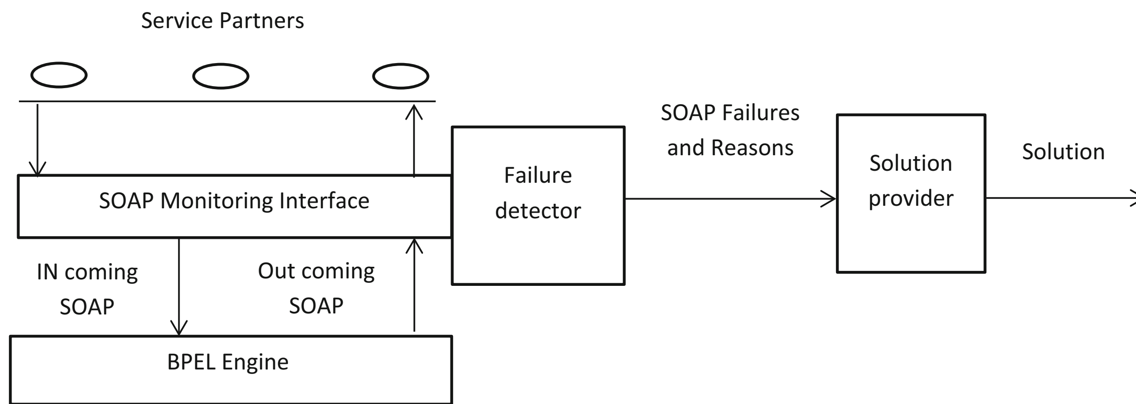

Fig. 4 Structure of FRWSC

of their recovery. The results of using the proposed framework were compared to other approaches using the same scenarios.

The evaluation of composite service against failure has been reported in the literature using different methods. Nik Looker et al. in [44] evaluated their work to avoid service failure in distributed system by creating a fault injector. The fault injector inserts errors in SOAP messages. Then, they measure the possibility of their work to overcome different failures. Omid Bushehrian et al. in [45] also used error injection method to evaluate their work at the design level. They evaluated the composite service workflow by assuming different individual service failures. Vadym Borovskiy et al. in [46] presented a solution to avoid failure of service invocation when the service changes its interface in WSDL file. They used the ".NET" framework to compare between the old and new versions of WSDL file.

The empirical evaluation of the proposed work measures the ability of the proposed framework to detect and avoid composite service failure. Different failure scenarios were traced during the execution of the composite service. The results are based on fulfillment of the requirements listed in Sect. 3. The evaluation process compares between the traditional design of composite service and the enhanced design using FRWSC. A set of metrics were chosen to measure the flexibility improvement of the composite service after using FRWSC features. The evaluation was performed at both the design level and the coding level. Evaluating the framework at the coding allows tracing the composite service at runtime. However, the evaluation at the design level gives clearer vision of the structure of the framework [47].

7.1 Evaluation perspective

The proposed framework can be evaluated from different perspectives such as performance, security aspects, and failure protection. However, the main goal of the framework is to maintain composite service execution and manage failures in atomic services. Thus, the evaluation process is concerned with error detection and failure recovery. Error detection and failure recovery process is broken into the following subprocesses.

1. Detecting errors in the SOAP message sent from the client-side interface.
2. Calculating the response time of the service provider and comparing it with the response time in the QoS attributes of the invoked service.
3. Detecting errors in the SOAP messages sent back from the provider to the client.
4. Extracting the failure reasons from the SOAP response messages.
5. Taking the decision of when and how to replace one service with another.

7.2 Evaluation process

The evaluation process has been designed to trace different atomic service failures to inspect if they can be avoided using the proposed framework. Hence, the aim of the evaluation process is to prove that the framework can overcome failures that traditional composite service cannot detect or overcome. A "Travel planner" composite Web service has been created and different failures were injected during execution. "Travel planner" aggregates different services such as flight booking, travel insurance, hotel reservation, and car rental to arrange a trip. "Travel planner" aggregates different services from different separated organizations. Each service has its own functionality that is separate from other services. However, when the services are aggregated together, different composition dependencies appear. The proposed work divides the composition dependencies between services into control flow, data flow, and precondition dependency. Four different designs were created to compare the proposed framework

with other approaches that protect composite service from failures. The comparison was made at both the design level and implementation level.

The first design is the traditional service composition using BPEL. The second is the same composite service after applying FRWSC features. The third design applies the variability approach [9,10,24], and the fourth applies service replication approaches to avoid failure [11,24]. The evaluation process is carried out by tracing the design and execution of a composite service that aggregates different services to arrange a journey for a client. The client provides the composite service with his name, address, telephone number, payment, destination, and the traveling date. Atomic services are built using Java or invoked from "Xmethod" repository. The services were built using Java on Eclipse environment and compiled using Java development toolkit 7 (JDK7). Then Apache Tomcat Server has been used to transform the Java code to Web services. Each Java class has been converted to an operation in a Web service. Using the apache server a local host server has been created at port 8080 to act as a server provider. Then a SOAP client-side interface has been created to accept the client data required to establish the invocation process. Performing the evaluation process as mentioned before is based on the following reasons.

1. The proposed method supports monitoring SOAP request of the client messages and extracting the client parameters to check the type and value of these parameters.
2. It is possible to calculate the exact time between sending the SOAP request and receiving the SOAP response from the provider.
3. It is possible to monitor the SOAP response from the provider and extract the implicit and explicit errors.

After the implementation of the atomic service, the BPEL engine has been used to orchestrate the composite service. The reason behind using BPEL is that it enables the description of business process in terms of message exchange between collaborating services. In addition, BPEL is an XML document that allows parameter exchange between services in SOAP form. This feature gives BPEL an advantage over other programing languages. Traditional programming languages need to preform SOAP encoding and decoding process when parameters are exchanged between services. Thus BPEL requires less computational overhead and has higher performance.

Different scenarios have been used to trace the composite service. Each scenario represents a failure reason from those discussed in Sect. 4. Then, the behavior of the composite service is measured according to different metrics listed in Sect. 7.3. The following cases have been used.

- Traditional composite service
- Composite service supported against failure with the proposed framework
- Composite service enriched with variability approach and service replication approach.

7.3 Evaluation metrics

The criteria to choose the evaluation metrics are based on the ability of each metric to measure the flexibility improvement to composite services. The chosen metrics should demonstrate the difference between the traditional and the FRWSC based composite service in dealing with errors. In addition, the chosen metric should differentiate between different kinds of errors according to their frequency of occurrence. For instance, the errors that occur frequently should have a higher weight than errors that rarely occur. Following is the evaluation metrics list that satisfies the previous requirements.

- The number of errors that can be avoided. Different errors should not be treated the same. Some error types occur frequently compared to others.
- The number of failures that can be repaired.
- The number of failures that the proposed work can overcome.
- The degree to which it is possible to preserve the functionality of the composite service after the process of service replacement.
- The impact on the performance of the composite service execution.

7.4 Evaluation results

The composite service of "Travel planner" is orchestrated using the BPEL engine under Eclipse. Then, the Orchestration Device Engine (ODE) has been used to aggregate service orchestration from different providers. Figure 5 shows the difference between traditional design of the composite service and the design supported by FRWSC. FRWSC is added to the composite service design to support it with the sufficient flexibility to respond to errors during runtime. Then, different types of error scenarios were inserted at different places during the composite service execution. The error scenarios are related to the client side, the provider side, and the communication between them. The following list presents the errors which are detected and identified by applying FRWSC to the composite service.

1. Wrong type of input parameters
2. Wrong number of input parameters
3. Application layer communication errors
4. Transport layer communication error
5. QoS response time violation

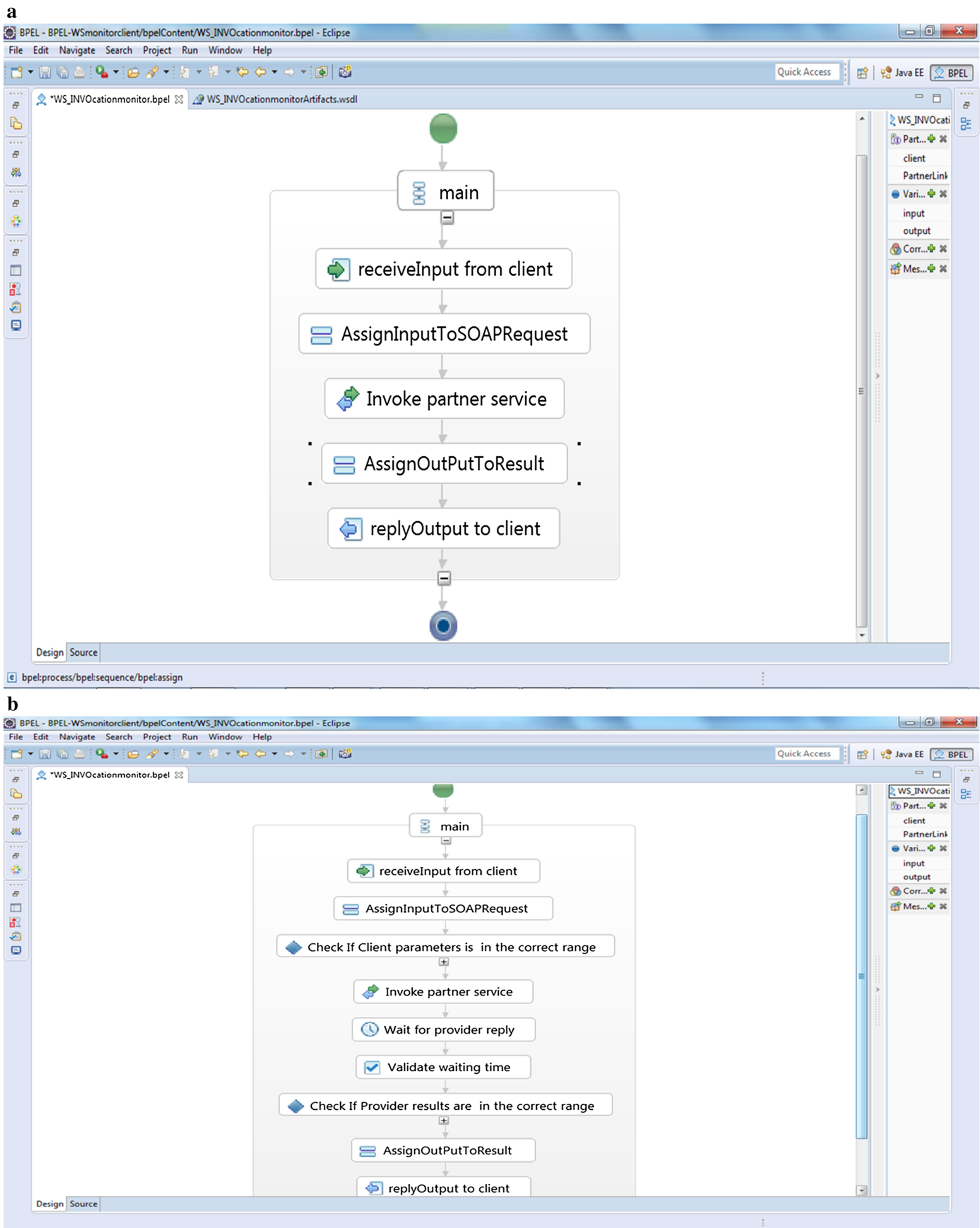

Fig. 5 **a** Snapshot of traditional design of a composite service, **b** snapshot of workflow design of composite service supported with FRWSC features

6. Bugs in provider service that result in parameters out of the range.
7. Time out of provider server

The same experiment has been performed using both the variability approach and replication approach. The traditional composite service design is also traced against the same errors and the results are listed in Table 5.

The errors listed in Table 6 cannot be detected by any of the approaches listed in the related work. Software variability and service replication treat all errors in the same way. These approaches cannot differentiate between different failure reasons. To prove the ability of FRWSC to deal with wrong parameters in client input and provider result, "Travel planning" was traced in two different scenarios. In the first scenario, a wrong input parameter was inserted. The inserted input is invalid destination town name. The variability approach stopped execution at the failed variant point and invoked another service. The new service returned the same error as the client input has not been changed. The service replication approach invoked the same service from another location. However, this action did not repair the failure because invoking the same service with the same wrong parameters will return the same error. By applying such wrong parameter to FRWSC, the results showed that it detected the wrong parameter. Then, a message was returned to the client indicating the error. Moreover, FRWSC stops the composite service execution and saves all the client data.

The second scenario focuses on tracing "Travel planner" service by invoking an atomic service including code bugs. The variability approach invoked another service and repaired the failure. Service replication approach invoked the same service from another location and the failure was not recovered. However FRWSC was capable to recognize and identify the failure reasons. FRWSC took the right decision to replace the service with an identical one. Hence, the composite service was protected from failure and continued its execution safely.

7.5 Runtime evaluation

This subsection evaluates the framework from failure recovery perspective.

7.5.1 Detecting atomic service failure

To prove that FRWSC is able to deal with various types of failures at runtime, different composite services failures have been analyzed with and without the framework support. The evaluation criteria were chosen to prove the ability of FRWSC to detect, report, and overcome different failure reasons. The evaluation process demonstrates different failure scenarios that could occur during service invocation due to different

Table 5 Composite service response to different error types using different designs of failure protection

Design approach	Traditional composite service		Variability		Service replication		FRWSC	
Error types	Error message	Service status	Error message	Service status	Error message	Service status	Error message	Service status
Client input data (wrong type of input parameters)	HTTP error	Failed	HTTP error	Failed	HTTP error	Failed	Detailed message with wrong input	Automated recovery
Client input data (wrong number of input parameters)	No error	Failed	No error	Failed	No error	Failed	Detailed message with wrong input number	Automated recovery
Communication between service client and service provider	HTTP error Or no response	Failed	HTTP error or no response	Recovery is possible	HTTP error or no response	Recovery is possible	HTTP error or time out	Automated recovery
Provider-side error (Bugs in provider service)	Wrong results	Failed	Wrong results	Recovery is possible	Wrong results	Failed	Detailed reason of provider	Automated recovery
Provider-side error (Time out of provider server)	No response	Failed	No response	Recovery is possible	No response	Failed	Detailed reason of provider time out	Automated recovery

Table 6 Action taken by different approaches to avoid composite service failure

Design approach	Variability		Service replication		FRWSC	
Error types	Error explanation	Action taken	Error explanation	Action taken	Error explanation	Action taken
Client input data (wrong type of input parameters)	No failure explanation	Another service will be invoked with the same wrong parameters	No failure explanation	The same service will be invoked from another location with the same wrong parameters	Detailed message with wrong input	The client input is corrected and the failure is recovered
Provider-side error (Bugs in provider service)	No failure explanation	Another service will be invoked Recovery is possible	No failure explanation	The same service will be invoked from another location Resulting in the same failure	Detailed reason of provider error	Another service from selected services will be invoked to recover the error

reasons. Then, the same failed service is invoked again with the same failure reason but after applying the framework. The results proved that the FRWSC was capable to detect SOAP errors that were not detected by traditional invocation methods. The proposed framework has been implemented as an interface between the composite service and its atomic services. FRWSC monitors the SOAP messages between the composite service and different individual services. Thus, different errors could be detected and identified.

FRWSC has been evaluated against three failure reasons

1. Service failure due to wrong client input parameters.
2. Service failure due to mismatch in XML schema definition.
3. Service failure due to errors in the provider software.

1. Service failure due to wrong client input parameters

Wrong input parameters can be a reason for Web service failure [6]. Such failure reason is not reported to the user as mentioned before in Sect. 3. Section 3 demonstrated a case for a service which calculates the temperature of the Mediterranean capitals. The service accepts the double code of the country and returns the temperature of its capital. If the service is invoked with wrong parameters, the SOAP reply will not include any error. Such situation means that neither the client nor the composed service will detect that this invocation has failed. In this scenario, the string "We" was used as input. "We" is not a legal double code of any Mediterranean country. The reply is "0" and no error message is reported to the user and such value could be passed to another service. To evaluate the proposed framework, the same service has been invoked with the same error but after applying the framework. FRWSC monitored the out coming SOAP messages and checked its parameters to detect whether it is in the permissible range or not. The framework interrupted the requested message including "We" and recognized that it is not a legal input. Then, it reported an error message to the client advertising him to change the wrong input with a correct one. Figure 6a shows the SOAP messages between the client and the provider without using FRWSC. While Fig. 6b shows the SOAP request and response after applying FRWSC. Figure 9b shows that FRWSC is able to detect and overcome service failure due to wrong input parameter.

2. Service failure due to mismatching in XML schema definition.

The second failure reason that can be avoided and reported using FRWSC is the incompatibility of different XML schema definition. The user input parameters could have legal value, but due to different schema definition, type mismatch occurs between user parameters and the provider service. To give an example of such situation, the same temperature service was implemented using different environments. The

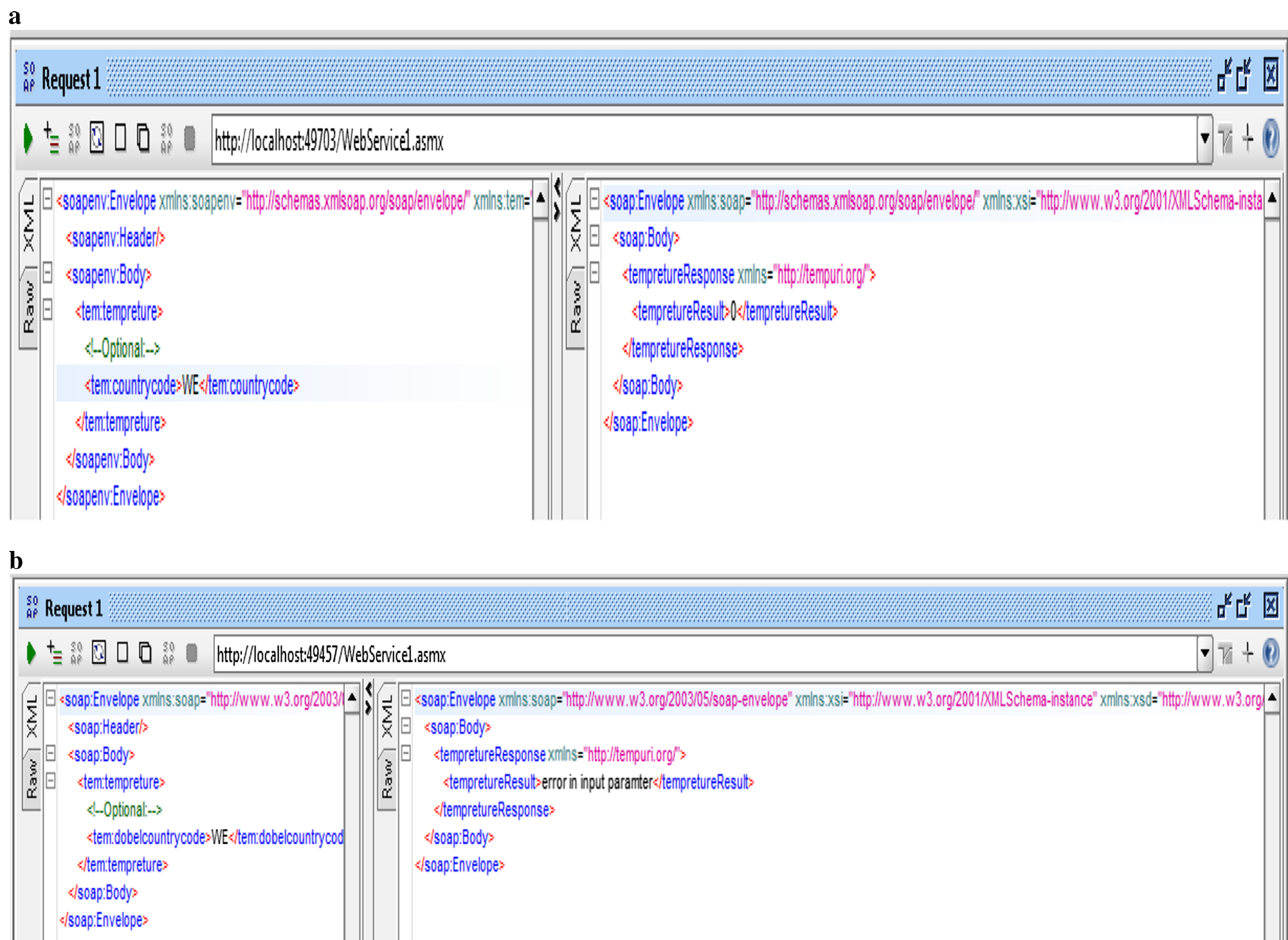

Fig. 6 **a** Snapshot for SOAP messaging including the error without FRWSC, **b** Snapshot for SOAP messaging including the same error after applying FRWSC

service in Sect. 3 was implemented using .Net framework and ASP.Net to build the client-side interface. The same service functionality was built using Java Eclipse, and then, Apache Tomcat server was used to create the Web service. Then, we created a local host at port 8080 which was used to acts as a service provider. The service was invoked using correct parameter (a double code of Mediterranean country). The service failed to return the correct result. By applying the proposed framework during service invocation, the framework monitored the SOAP request. It recognized that the parameters in the SOAP request are correct. However, while monitoring the SOAP reply, the framework detected the error. The solution provider which is enriched with the WSDL file of the invoked service detected the mismatch between the input parameter and the WSDL file. Hence, it advices the client to replace this service by another one. Listing 1 shows the difference between the WSDL files of both service implementations (using .Net framework and using Java Eclipse). Figure 7a shows the SOAP request and reply for the service built using .Net framework. The service was invoked using “IT” as input which represents the double code of Italy. The service returned the correct result. Figure 7b shows the SOAP messaging for the service which is implemented using Java. The same input “IT” was used, but the SOAP reply is not correct. The reason for such error is the difference in schema definition which defines the input string that is passed to the service. If such service had been invoked without using FRWSC, it would have not been possible to detect the error. However, the result shown in Fig. 10 proves that FRWSC is able to detect service failure as a result of mismatch in XML schema. The failed service has been replaced.

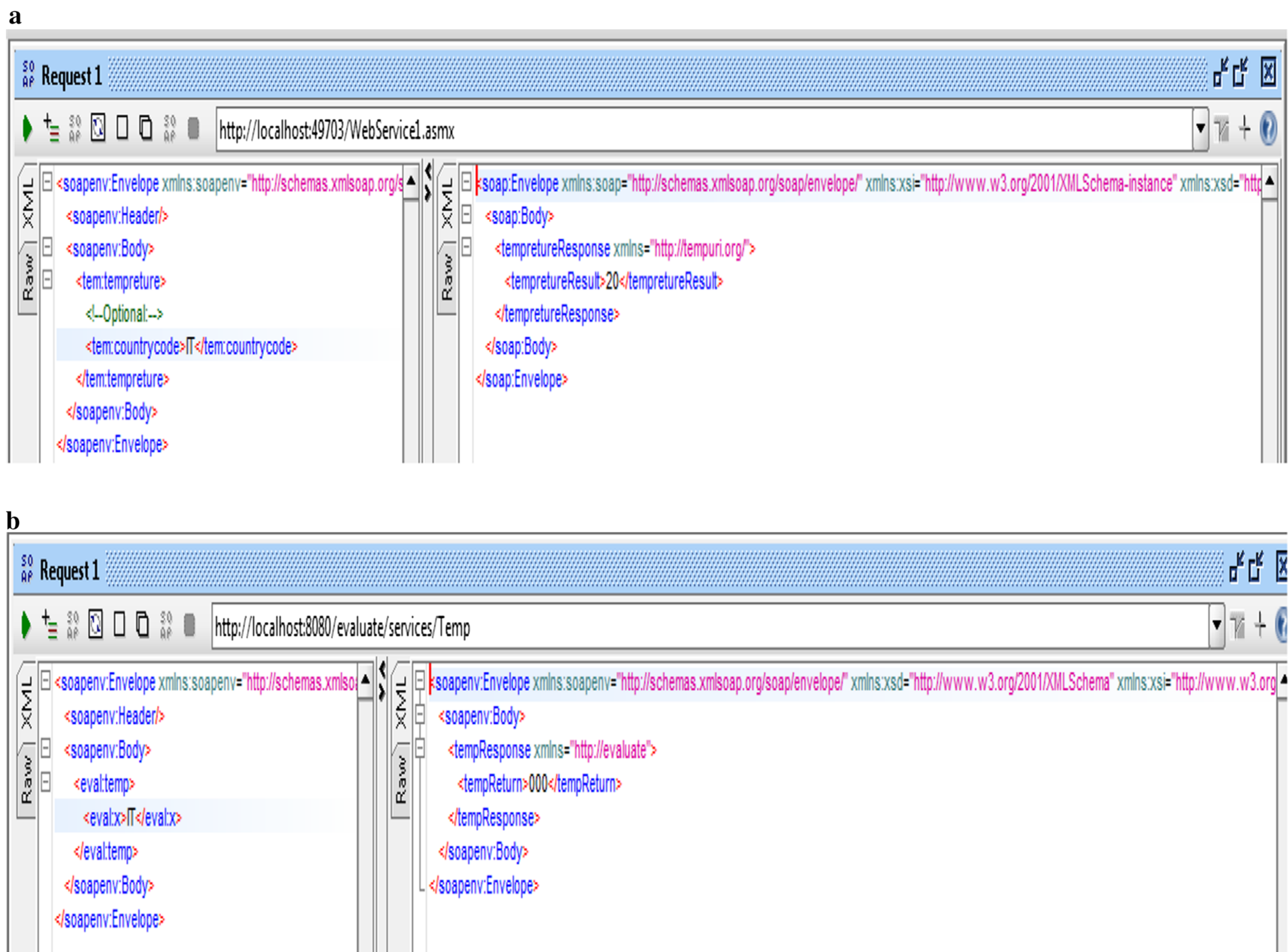

Fig. 7 **a** Snapshot for SOAP request and response for a successful invocation with user input "IT," **b** snapshot for SOAP request and response for a failed invocation using same input "IT"

Listing 1.a WSDL file for temperature service built using .Net frameworks

Listing 1.b WSDL file for the temperature service built using Java Eclipse

```
<wsdl:definitions targetNamespace="http://tempuri.org/">
<wsdl:types>
<s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
<s:element name="tempreture">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="countrycode" type="s:string"/>
</s:sequence>
</s:complexType>
</s:element>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://evaluate" xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://evaluate" xmlns:intf="http://evaluate" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<wsdl:types>
<schema elementFormDefault="qualified" targetNamespace="http://evaluate"
xmlns="http://www.w3.org/2001/XMLSchema">
<element name="temp">
<complexType>
<sequence>
 <element name="countrycode" type="xsd:string"/>
</sequence>
```

3. Service failure due to errors in the provider software.

When a client invokes a composite service, he gets back the final output. The problems of the individual services within the composite service are not reported to the client. The failure reason could be due to a bug in the provider software. To illustrate such situation, the temperature service was invoked using correct input, but due to a possible software bug, the provider software returned the value 88. This result is not a valid temperature for any city. If this value is passed as input to another service, any other calculation will be done with such wrong input.

By applying FRWSC, it monitored the SOAP message including the provider reply. It checked the value of the invocation result and detected that it is out of the permissible range. So it returned an error message to the client with the failure and its reason. Figure 8a shows a legal invocation using the string "FR" which is the double code of France. The SOAP reply returned "88" which is a wrong result due to a software bug in the provider code. Such error cannot be detected and this value could have been be passed to another service as input. At the end of this scenario, the composite service would return a wrong result without knowing which service is responsible for the error. Figure 8b shows the same service invoked by the same parameter "FR" but after applying FRWSC. The framework detected the error in the reply message and reported an error message to the client. Hence, FRWSC is able to detect failures that results from bugs in provider code.

7.5.2 Composite service failure due to mismatching between successive services

Although the evaluation at design level gives a better vision of the evaluated system, the FRWSC is evaluated and traced during execution to proof the validation of the proposed concept. After designing the composite service using BPEL Eclipse, an ODE engine was created to manage the orchestration of different service partners. The ODE was used to create the executable WSDL file from the XML based BPEL file. Then, the WSDL file was tested during execution with "Web service explorer" which allows monitoring the composite service during execution. The communication between the composite service and its constituent services was performed via SOAP messages. In several situations, the parameters included in the SOAP response of one service had to be transferred to the SOAP request of the successive service.

In the traced scenario, a partner service sends its response parameters including a variable in string type. Then, these parameters were passed to another partner service that accepts the input in integer type. This situation was traced during the execution and resulted in the error shown in Fig. 9. Figure 9 shows a snapshot of testing the composite service execution in which there are errors at the server side or at the HTTP communication. The lower side of the figure shows that the server has started successfully. At port 8080 of the local host, the communication layer has no errors. The real error occurred because of the mismatch between the output of a service and the input of the successive one. The returned message does not clearly identify the error so the client could not repair it. When such situation is applied to the FRWSC, the returned message clearly identifies the error and the solution provider can swap the second invoked service with a service that accepts string input type.

7.6 Average recovery time

This section evaluates the mechanism of service replacement. "Mediterranean towns temperature" service was used to prove the ability of FRWSC to choose alternative services and to measure the time taken to recover the failure. A service including software bugs was invoked, and its result and execution time were calculated using SOAP-UI. The failed service returned a temperature that equals 188 for the capital of France. The invocation duration took 29 msec. Figure 10a shows the failed service and its execution time. Then, after applying FRWSC the same service was invoked again. The results show that FRWSC was able to detect the error and then replace the failed service with another service. The cor-

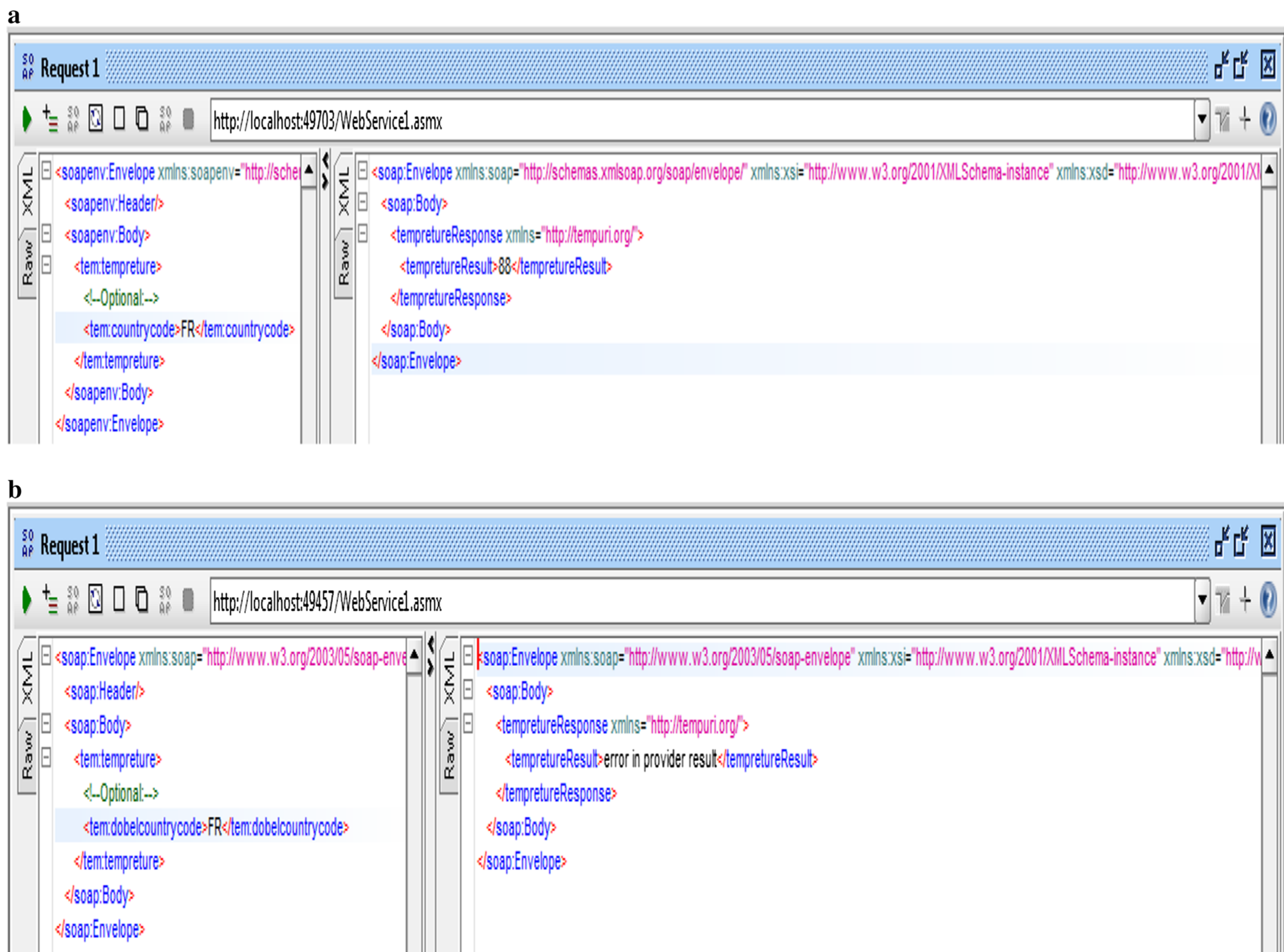

Fig. 8 **a** Snapshot for SOAP request and respond including illegal result without error, **b** Snapshot for SOAP request with same parameters after applying FRWSC

rect results were sent back to the user. The time to detect the error and replace the failed service with another was calculated. Figure 10b shows that it took 316 msec to detect and recover the failed service. The recovery time took 287 msec. This proves that FRWSC is able to choose an alternative service.

7.7 Limitation of FRWSC

The ability of the presented solution to recover service failures due to wrong parameters is limited to the values that are stored in the I/O parameter range. Network failures can be detected at the application layers; hence, FRWSC cannot distinguish between different network layers failures. Also the capability of FRWSC to detect and recover errors is limited to the hardware on which FRWSC is running and the reliability of its different services. For instance, “BPEL and SOAP monitoring” service, “checking parameter” service, and “service replacement” service.

7.8 Comparing FRWSC with industrial solutions

This section comprise between FRWSC and ESB (Enterprise service Bus). ESB is an industrial method concerns with composite service reliability that supports service interaction monitoring. ESB focuses on routing clients’ requests to the appropriate services while this work is concerned with composite service failure protection. ESB defines a model for a standard set of messages that allow ESB to communicate with different services [50]. A software adaptor is used to fulfill the transformation of service messaging to ESB messages. This results in communications overhead. In addition, ESB requires providers to integrate ESB into their services to be published in ESB registries [50]. Such mechanism means that ESB is not for general and open use. ESB cannot detect wrong input parameters of the client requests or the provider results. It cannot provide a solution if no service is found to fulfill the client request. FRWSC can be used in general services registries without adding new message format to service communication. FRWSC can detect and avoid fail-

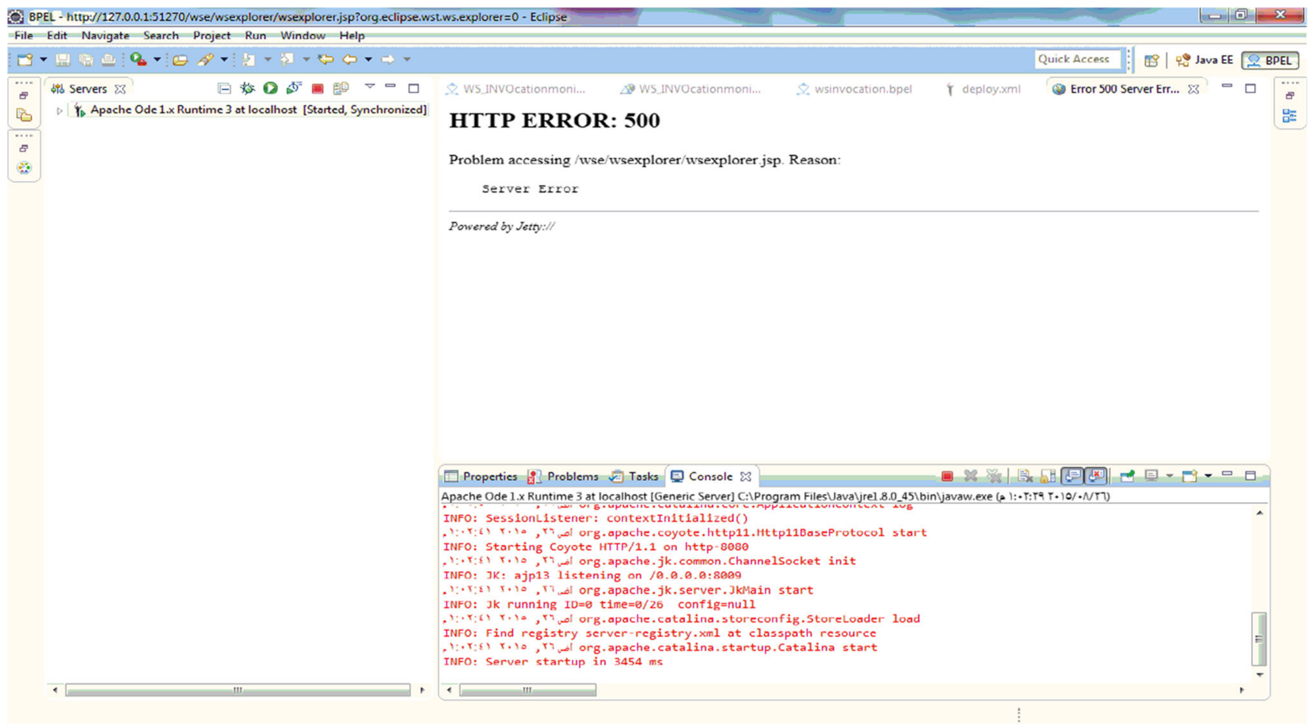

Fig. 9 Error in WSDL execution due to type mismatching between partner services

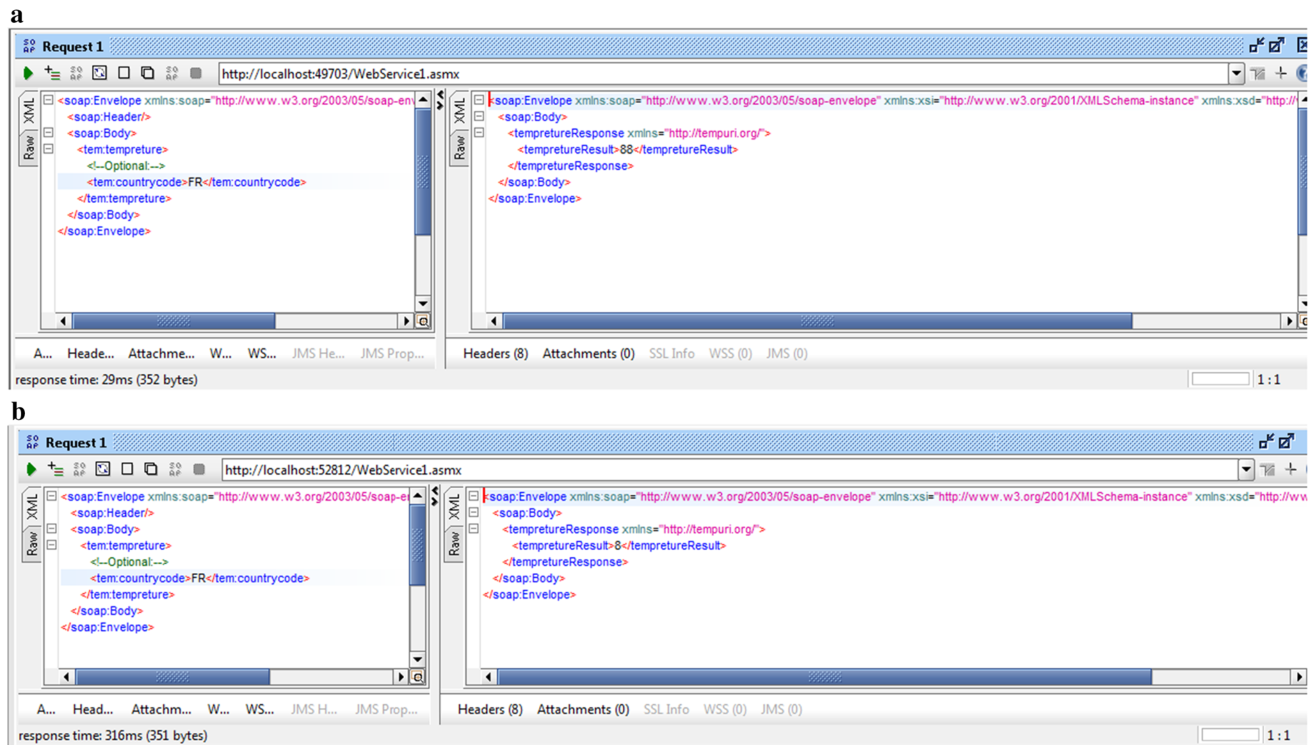

Fig. 10 **a** Time taken to execute a wrong service invocation, **b** time taken to replace the failed service with a correct one

ures due to wrong parameters in client request and provider results. FRWSC contains a solution provider that can roll back to AI planner to divide an individual task into smaller tasks if there is no service that can fulfill the required task.

8 Conclusion and future work

This work addressed a crucial problem in service composition concerning the limitation of SOAP error detection and reporting capability. By tracing the execution of composite services, we concluded that SOAP error messages cannot provide the Web service client with the sufficient information about service failure. We presented a framework to address the problem of failure in composite services as a result of failure or change in atomic services. The presented work deals with failures resulting from incorrect user parameters, bad network connections, and provider-side errors. The proposed framework identifies the failure reason and returns a clear message describing it to the client to take the correct decision during the execution of the composite service. We evaluated the work against failure protection approaches. The results proved that using FRWSC, it is possible to avoid different kinds of composite service failures that cannot be avoided by traditional methods. We believe that the presented work will enhance the reliability of Web service composition and consequently increase its usage. Our future work includes enhancing SOAP performance and building exception handlers for Web service invocation.

References

1. Curbera F, Duftler M, Khalaf R, Nagy W, Mukhi N, Weerawarana Sanjiva (2006) Unraveling the web services : An introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput 6(2):86–93
2. Nama W, Kilb H, Leecthe D (2008) On the computational complexity of behavioral description-based web service composition. In: 20th IEEE international conference on tools with artificial intelligence (ICTAI), 2008
3. Sung-Shik T, Jongmans Q, Santini F, Sargolzaei M, Arbab F, Afsarmanesh H (2014) Orchestrating web services using Reo: from circuits and behaviors to automatically generated code. Serv Oriented Comput Appl 8(4):277–297
4. Xu Bin, Luo Sen, Yan Yixin, Sun Kewu (2009) Towards efficiency of QoS-driven semantic web service composition for large-scale service-oriented systems: In Service Oriented Computing and Applications journal Volume 6, Issue 1, pp 1-13, Springer
5. Canfora G, Di Penta M, Esposito R, Villani ML (2008) A framework for QoS-aware binding and re-binding of composite web services. J Syst Softw 81(10):1754–1769
6. Bushehrian Omid, Zare Salman (2012) Navid Keihani Rad. A Workflow-Based Failure Recovery in Web Services Composition: Journal of Software Engineering and Applications 5:89–95
7. Web Services Description Language, WSDL (2014) Web site at: http://www.w3.org/TR/wsdl, 2014
8. The SOAP specification(2014) by the World Wide Web Consortium, available on-line at http://www.w3.org/TR/SOAP/, 2014
9. Sun Chang-ai, Rossing Rowan, Sinnema Marco, Bulanov Pavel, Aiello Marco (2010) Modeling and managing the variability of Web service-based systems. J Syst Softw 83:502–516
10. Koning M, Sun C, Sinnema M, Avgeriou P (2009) VxBPEL: supporting variability for Web services in BPEL. J Inf Softw Technol 51:258–269
11. Liu A, Li Q, Huang L, Xiao M (2010) FACTS: a framework for fault-tolerant composition of transactional web services. IEEE Trans Services Comput 3:46–59
12. Zheng Z, Zhang Y, Lyu MR (2010) Distributed QoS evaluation for real-world web services. In: IEEE international conference on Web services, 2010
13. Salas J, Perez-Sorrosal F, Patiño-Martinez M, Jimenez-Peris R (2006) WS replication :a framework for highly available web services. In: World wide web conference committee (IW3C2) Edinburgh, Scotland ACM 1-59593-323-9/06/0005
14. Zheng Z, Lyu MR (2008) A distributed replication strategy evaluation and selection framework for fault tolerant web services. In: Proc. 6th international conference of web services(ICWS'08), p 145–152
15. Yu T, Zhang Y, Lin K-J (2007) Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans Web 1(1):1–26
16. Curbera F, Duftler M, Khalaf R, Nagy W, Mukhi N, Weerawarana Sanjiva (2006) Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput 6(2):86–93
17. Senagi KM, Okeyo G, Cheruiyot W, Kimwele M (2015) An aggregated technique for optimization of SOAP performance in communication in Web services. Serv Oriented Comput Appl 9, doi:10.1007/s11761-015-0186-x, Springer
18. Le'cue' F, Mehandjiev N (2011) Seeking quality of web service composition in a semantic dimension. IEEE Trans Knowl Data Eng 23(6):942–959
19. Batra U, Mukherjee S (2011) Enterprise application integration (Middleware): Integrating stovepipe applications of varied enterprises in distributed middleware with service oriented architecture. In: IEEE international conference on electronics computer technology, 2011
20. Fu X, Bultan T, Su J (2005) Synchronizability of conversations among web services. IEEE Trans Softw Eng 31(12):1042–1055
21. Baresi L, Guinea S (2005) Towards dynamic monitoring of WS-BPEL processes. In: Benatallah B, Casati F, Traverso P (eds) Service-oriented computing—ICSOC 2005. Lecture Notes in Computer Science, Vol. 3826, Springer, Berlin, pp 269–282. doi:10.1007/11596141_21. ISBN:978-3-540-30817-1
22. Uddin MS, Ripon S, Das NC, Hossain O (2014) A comparative study of web service composition via BPEL and petri nets. Int J Comput Elect Eng 6(2):110
23. Hatzi O, Vrakas D, Nikolaidou M, Bassiliades N, Anagnostopoulos D, Vlahavas I (2010) An integrated approach to automated semantic web service composition through planning. IEEE Trans Serv Comput TSC-2010-06-0088.R2
24. Bruneo D, Distefano S, Longo F, Scarpa M (2013) Stochastic evaluation of QoS in service-based systems. IEEE Trans Parallel Distrib Syst 24(10):2090–2099
25. Scanniello G, Gravino C, Genero M, Cruz-Lemus J, Tortora G (2014) On the impact of uml analysis models on source-code comprehensibility and modifiability. ACM Trans Softw Eng Methodol 23(2) Article 13, Pub. date: March 2014
26. Guinard D, Trifa V, Karnouskos S, Spiess P, Savio D (2010) Interacting with the SOA-Based Internet of Things: Discovery, query, selection, and on-Demand Provisioning of Web Services. In Serv Comput IEEE Trans 3(3):223–235
27. Rosario S, Benveniste A, Haar S, Jard C (2006) Foundations for web services orchestrations: functional and QoS aspects. In: ISOLA '06: Proceedings of the 2nd international symposium on

leveraging applications of formal methods, verification and validation, Washington, DC, USA: IEEE Computer Society, 2006, pp.309–316
28. Arenas M, Daenen J, Neven F, Van den Bussche J, Ugarte M, Vansummeren S (2014) Discovering XSD keys from XML data. ACM Trans Database Syst 39(4) Article 28, Publication date: December 2014
29. Lo W, Yin J, Deng S, Li Y, Wu Z, Collaborative (2012) Web Service QoS prediction with location-based regularization. In: Web Services (ICWS), IEEE 19th international conference on, pp 464–471. IEEE, 2012
30. D'Ambrogio A, Bocciarelli P (2007) A model-driven approach to describe and predict the performance of composite services. In: WOSP '07 Proceedings of the 6th international workshop on Software and performance. New York, NY, USA: ACM, 2007, pp. 78–89
31. Jiang J, Syst T (2005) UML-based modeling and validity checking of web service descriptions. In: Proceeding of IEEE international conference on web services, pp. 453-460, 2005
32. He J, Zhang Y, Huang G, Cao J (2012) A smart Web service based on the context of things. ACM Trans Internet Technol (TOIT) 11(3):13:1–13:23
33. Perepletchikov M, Ryan C (2011) The impact of service cohesion on the analyzability of service-oriented software. IEEE Trans Softw Eng 37(4):449–465
34. Nagamouttou D, Egambaram I, Krishnan M, Narasingam P (2015) A verification strategy for web services composition using enhanced stacked automata model. SpringerPlus. doi:10.1186/s40064-015-08051
35. Zhang W, Sun H, Liu X, Guo X (2014) Temporal QoS-aware web service recommendation via non-negative tensor factorization. In: World wide web conference committee (IW3C2), Seoul, Korea, ACM 978-1-4503-2744-2/14/04 April 7–11, 2014
36. Erradi A, Maheshwari P, Tosic V (2006) Recovery policies for enhancing web services reliability. In: IEEE international conference on web services (ICWS'06) 0-7695-2669-1/06
37. Sun J-T, Zeng H-J, Liu H, Lu Y, Chen Z (2005) Cubesvd: a novel approach to personalized web search. In: Proceedings of the 14th international conference on World Wide Web, pp 382–39, ACM, 2005
38. Erradi A, Maheshwari P (2005) QoS-aware middleware for reliable web services interactions. In: IEEE international conference on e-technology, e-commerce and e-service (EEE'05), Hong Kong, 2005
39. Fu X, Bultan T, Su J (2004) Analysis of interacting BPEL web services. In: Proc. of the 13th International World Wide Web Conference (WWW 2004), New York, NY, USA, pp. 621–630, May 2004
40. Borger E, Fleischmann A (2015) Abstract state machine nets: closing the gap between business process models and their implementation. In: Proceeding of the 7th international conference on subject-oriented business management ACM New York, NY, USA
41. Anfeng L, Zhigang C, Hui H, Weihua G (2007) Treenet: a web services composition model based on spanning tree. In: Proceedings of the 2nd international conference on pervasive computing and applications, (ICPCA 2007), IEEE
42. Karunamurthy R, Khendek F, Glitho RH (2006) A novel business model for web service composition. In: Proceedings of the IEEE international conference on services computing (SCC'06), IEEE
43. Vukovi'c M, Kotsovinos E, Robinson P (2007) An architecture for rapid, on-demand service composition. J Serv Oriented Comput Appl-SOCA 1:197–212
44. Looker N, Xu J (2004) Assessing the dependability of SOAP RPC-based web services by fault injection. In: Proceedings of the 9th IEEE international workshop on object-oriented real-time dependable systems (WORDS'03) 0-7695-2054-5
45. Bushehrian Omid, Zare Salman, Keihani Navid (2012) A Workflow-Based Failure Recovery in Web Services Composition. Journal of Software Engineering and Applications 5:89–95
46. Borovskiy V, Zeier A, Karstens J, Ulrich H (2013) Resolving incompatibility during the evolution of web services with message conversion. In: Proceedings of the 3rd international conference on software and data technologies—SE/GSDCA/MUSE, pp 152–158. doi:10.5220/0001880101520158
47. Lindvall Mikael, Tvedt Roseanne Tesoriero, Costa Patricia (2003) An empirically-based process for software architecture evaluation. Empir Softw Eng 8:83–108
48. Behl J, Distler T, Heisig F, Kapitza R, Braunschweig TU, Schunter M (2012) Providing Fault-tolerant execution of web-service–based workflows within clouds. In: Cloud CP 2012: 2nd International Workshop on Cloud Computing Platforms, Bern, Switzerland, ACM 978-1-4503-1161-8
49. Hu J, Guo C, Wang H, Zou P (2005) Web services peer-to-peer discovery service for automated web service composition. The book networking and mobile computing, Vol. 3619 of the series Lecture Notes in Computer Science pp 509–518
50. Schmidt M-T, Hutchison B, Lambros P, Phippen R (2005): The enterprise service bus: making service-oriented architecture real. IBM Syst J 44(4):509