Predictive Failure Recovery in Constraint-aware Web Service Composition

Touraj Laleh, Joey Paquet, Serguei Mokhov and Yuhong Yan
Concordia University, Montreal, Quebec, Canada

Keywords: Web Service Composition, Service Constraints, Service Failure Recovery.

Abstract: A large number of web service composition methods have been proposed. Most of them are based on the matching of input/output and QoS parameters. However, most services in the real world have conditions or restrictions that are imposed by their providers. These condition should be met to ensure the correct execution of the service. Therefore, constraint-aware service composition methods are proposed to take care of constraints both at composition and execution time. Failure to meet constraints inside a composite plan results in the failure of execution of the whole composite service. Recovery from such failures implies service usage rollback as an alternate plan is found to continue the execution to completion. In this paper, a constraint-aware failure recovery approach is proposed to predict failures inside a composite service. Then, a method is proposed to do failure recovery based on those predictions and minimize the number of service rollbacks. The proposed solution includes an AI-planning-based algorithm and a novel constraint processing method for service failure prediction and recovery. A publicly available test set generator is used to evaluate and analyze the proposed solution.

1 INTRODUCTION

Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web (Rao and Su, 2005). Services can be registered and subsequently selected based on customer's requirements. However, if there is no specific web service that can fulfill a customer's requirements, there should be a possibility to combine existing services together to accomplish a user-specified task. Automatic web-service composition (AWSC) consists in the automated design of an appropriate combination of existing web services to achieve a global goal. A large number of approaches have been proposed to solve AWSC and most of them involve mixing and matching web services components according to their input, output and QoS features (Ponnekanti and Fox, 2002; Lécué and Léger, 2006; Berardi et al., 2003; Oh et al., 2008; Zheng and Yan, 2008; Hashemian and Mavaddat, 2005). There are limitations and preferences, called constraints, that must be considered to ensure correct execution of composite services. Such preferences and limitations which are specified by customers are called *customer constraints*. Furthermore, services have usage restrictions and QoS limitations (called *service constraints*) that are imposed by their providers.

For a composite service, the set of constraints are derived from the union of all constraints of the services that compose them. Whenever a composite service is executed, all its constraints should be verified to ensure its correct execution. Constraint verification for a composite service is different from constraint verification for a single service. The constraints of an individual service only have to be verified before its execution. However, constraints applied on a composite service can be verified *during* its execution, as each individual component service is executed. Indeed, the verification of some individual services constraints actually depend on the values that are to be provided by users or other services during the execution of a composite service. In this situation, if the restrictions that are set by these constraints will not be met at execution time, the service execution fails and consequently fails the execution of the composite service. This failure might result in service rollbacks if some already used services have to be replaced in order to successfully execute a composite service. For example, consider a composite shopping service including product search, payment and shipment services, where the shipment service can only ship products to specific areas in North America. During the execution of a shopping task, if the search service selects a product outside North America and

Laleh, T., Paquet, J., Mokhov, S. and Yan, Y.
Predictive Failure Recovery in Constraint-aware Web Service Composition.
DOI: 10.5220/0006313802410252
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 213-224
ISBN: 978-989-758-243-1

payment service makes a payment, the shipment service fails the execution of the whole composite service even though the order has already been paid before the shipment service fails. In this case, the execution of the payment service needs to be rolled back. This is an example that shows how service constraint failure can result in the failure of the execution of a composite service, where rollback is necessary in order to fix the situation.

Many failure recovery approaches have been proposed to manage and recover failure in the execution of composite services (Gao et al., 2011). Web service transaction (WST) approaches (Dolog et al., 2014; Xu et al., 2016; El Hadad et al., 2010) use recovery mechanisms, including forward and backward recovery. Forward recovery attempts to reach the original goal of the composite service by retrying or replacing components and continuing the process. Backward recovery is essentially a form of rollback that unrolls the transaction and tries to find another solution.

Some approaches consider component services constraints in web service composition (Wang et al., 2014; Wang et al., 2015). In these approaches, constraints are embedded inside a composite plan to be verified as the composite service is being executed. However, current failure recovery approaches do not consider individual service constraints.Therefore, these approaches are not useful for the execution of constraint-aware composite services because anytime a service is added or removed from a composite service, the set of constraints of the composite service needs to be updated accordingly. In addition, failure recovery approaches start recovery from the broken point in the plan and if the plan cannot be recovered from the broken point, the results of all services have been executed before the broken point should be rolled back (Dolog et al., 2014; Xu et al., 2016; El Hadad et al., 2010). Wang et. al (Wang et al., 2014) proposed a formal constraint-aware service composition method and introduced conditional branch structures into the process model of a solution. Then in (Wang et al., 2015), the uncertain effects of composite service execution are managed. In (Xu et al., 2016) a framework to improve and optimize the success rate of transactional composite services is proposed. (Dolog et al., 2014) introduced an approach to model compensation capabilities and requirements using a forward recovery approach and proposed a framework to reduce time and resource waste.
In this paper, a constraint-aware failure predication and recovery approach is proposed to to reduce the cost of service failures inside a composite plan using failure prediction. We provide a novel solution to assemble a *composite service package* including all possible solutions for a service composition problem.

There are approaches that have tried to address problems related to constraint verification, failure recovery and web service composition reliability. To the best of our knowledge, no approach provides a single solution for all the above problems. First, through some real-world business scenarios, we describe the problem. Second, a new constraint-aware composite service model is discussed. An algorithm to combine all possible solutions and create a composite service package is developed. Finally, two constraint-aware execution algorithms are developed to enable the execution of constraint-aware composite service packages. Section 1.1 describes a real world business scenario to show the issues involved. Section 1.2 discusses related work. Section 2 provides definitions for our composite service model and discusses research issues, and describes our proposed solutions for constraint-aware web service composition. Section 3 describes our approach to execute constraint-aware composite services, including recovery upon failure. A comparative experimental analysis of our approach versus other solutions is presented and discussed in Section 4. Finally, Section 5 concludes this paper.

1.1 Motivation Scenario

Consider a shopping application that consists of a set of tasks: searching for products, submitting an order, paying for the order, and shipping/delivery of the order. A customer application makes a request to the composition engine for a composite service that lets a user with a *DeliveryAddress* order a product (*ProductName*) and do the shipment. The user also specifies a constraint on the cost of the composite service. The available individual services are depicted in Table 1. For instance, the three shipping services have different applicable constraints, e.g., the standard shipping service (w_3) is available only for products whose delivery address is located in Montreal; two-day delivery (w_4) is available only for orders whose delivery address is located in the province of Quebec; while other shipping services (w_7 and w_8) are available for orders found in the other regions of Canada. Given the customer constraints and all respective service constraints, Figure 1 shows all possible composition plans that could fulfill the request from the customer.

Based on what we discussed in Section 1.1, there are three composition plans for the shopping service request that can accomplish the shopping task. In each plan, the shipment service has its

Table 1: Available Services.

#	Service	Input	Output	Cost	Constraints
W_1	Search	{ProductName, DeliveryAddress}	{ProductNumber, ProductAddress}	0.4	$C_1 = \emptyset$
W_2	Order/Payment	{ProductNumber}	{PaymentNumber, OrderNumber}	4	$C_2 = \emptyset$
W_3	Shipment	{PaymentNumber, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	2	C_{31} = DeliveryAddress ∈ Montreal C_{32} = ProductAddress ∈ Montreal
W_4	Shipment	{PaymentNumber, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	2.5	C_{41} = DeliveryAddress ∈ Quebec C_{42} = ProductAddress ∈ Quebec
W_5	Order	{ProductNumber}	{OrderNumber}	3	$C_5 = \emptyset$
W_6	Payment	{ProductNumber}	{PaymentConfirm}	3	$C_6 = \emptyset$
W_7	Shipment	{PaymentConfirm, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	1	C_{71} = DeliveryAddress ∈ Canada C_{72} = ProductAddress ∈ Canada
W_8	Shipment	{PaymentConfirm, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	10	C_{81} = ProductAddress ∈ Canada C_{82} = DeliveryAddress ∈ Canada

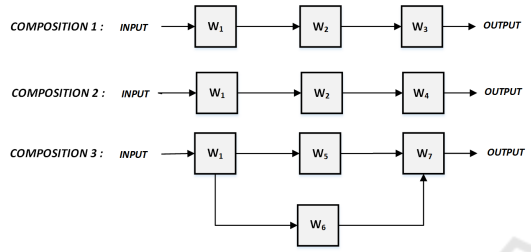

Figure 1: Possible composition plans.

own constraints based on the *ProductAddress* and *DeliveryAddress* of the orders. Consider that a user wants to use the composite service to buy a book. Given that the user specified cost optimization as a requirement, *composition* 1 (Figure 1) will be picked for shopping as the best composite service (minimum cost). During the execution of the composition plan, after searching the book (executing w_1), it may turn out that the product address is in Toronto. In this case, after ordering the product and making the payment (executing w_2), the execution of the shipment service (w_3) fails, as the product's address is not in Montreal. In this situation, w_1 and w_2 have already been executed and their executions results need to be rolled back.

This example shows that there are constraints (e.g. shipment constraint related to the delivery address) that can be verified before execution of the first service in the composition plan. However, some service constraints can only be verified during the execution (e.g. constraint related to product address) and their failure can fail the execution of the whole composite service. These sorts of failures cannot be predicted at composition time, as the verification depends on the values that are going to be produced during the execution of the composite service (e.g. *ProductAddress*). In this situation, a failure recovery approach is required to recover the plan. In (Xu et al., 2016; Gao et al., 2011) different dynamic failure recovery approaches are discussed including backward and forward recovery approaches. These approaches only start recovery in case a service failure happens. Backward recovery approaches (El Hadad et al., 2010) mostly need to rollback effects of executed services (w_1 and w_2) and find an alternative composite service to execute the task. However, as they do not consider service constraints, the alternative plans might also fail. For example, the best alternative plan (for composition 1) based on cost is *composition* 2, whose execution will fail as a result of the constraints imposed by w_4. Forward recovery based approaches (Dolog et al., 2014) look for an alternative service with the same functionality (input/output) to repair the plan (e.g. w_4 for w_3). However, as forward recovery approaches do not consider the constraints of alternative services, the recovered plans might fail again. Current recovery approaches do not consider service constraints and that could result in a recovered plan that fails again. They might find an alternative plan that executes the same services and fail the execution over and over or they might not even be able to recover the plan.

Having a constraint-aware composite service which is aware of its component services constrains during the execution can help to predict failure and avoid some wasted executions. For example, if the composite service execution system is aware of component services constraints during the whole execution process, it could check the shipment service constraint right after execution of the search service and avoid execution of payment service when the product address is not in the Montreal area. Therefore, the first issue is to design constraint-aware plans to be able to verify constraints more effectively to predict failures inside a composition plan. In addition, when failures are predicted, a constraint-aware failure recovery approach needs to be used to complete the task. Such failure recovery approach starts recovery sooner and minimizes rollbacks. As a result, the second issue will be designing a constraint-aware failure recovery mechanism to use failure prediction and start recovery as soon as a failure is predicted. Finally, the execution effects of some services are often uncertain because of the complex and dynamically changing application environments in the real world. This can cause different results in verification of the following constraints inside a composite service. Therefore, the third issue is to design an execution method that can verify constraints at run-time and make proper decisions for failure recovery. To the best of our knowledge, there is no constraint-aware failure recovery approach that can resolve all of the above-mentioned issues in which we are interested.

1.2 Literature Review

A considerable amount of work has been done in the theoretical modeling and practical implementation of web services. In this section, first we discuss service constraints and web service composition. Then, we review related work on composite service recovery approaches.

1.2.1 Web Service Composition and Constraints

Web Service composition researches are discussed into three different categories such as: formal methods-based approaches (Lécué and Léger, 2006; Berardi et al., 2003), AI planning techniques (Oh et al., 2008; Zheng and Yan, 2008) and graph-search-based approaches (Hashemian and Mavaddat, 2005; Wang et al., 2014; Brogi and Corfini, 2007).

Graph-based approaches usually construct a service dependency graph to show all possible dependencies based on input and output parameters. In most graph-based approaches the service dependency graph is a reflection of the underlying data interface relationships among services. In this context, AWSC acts like a graph search problem and finds a path either from provided inputs to required outputs or vice-versa. Most graph-based approaches do not consider service constraints and they model services based on input/output parameters. Hashemian et. al (Hashemian and Mavaddat, 2005) uses a modeling tool to convert the WSC problem into a general graph problem. Lang et. al (Liang and Su, 2005) present an AND/OR graph representation of search dependency graph and its search algorithm for the discovery of composite services. Wang et. al (Wang et al., 2014) propose a formal constraint-aware service composition method. The proposed solution includes a graph-search-based algorithm which generates all possible solutions. It then introduces conditional branch structures into the process model of a solution to solve the problems brought by service constraints, in case different services could accomplish the same task. However, this approach only considers the situation where different concrete services can be used for a sub task in a composition plan and does not discuss how to verify these constraints.

AI planning is another approach for the AWSC problem in which, given an initial state and a goal state, a sequence of actions can be acquired automatically through planning (Rao and Su, 2005). This approach is done in two stages: a forward expansion stage constructs a search graph and a backward searching stage retrieves a solution (Li et al., 2016). Some AI planning approaches (McIlraith and Son, 2002) address the web service composition problem through the provision of high-level generic procedures and customizing users constraints. Moreover, there are AI planning-based approaches (Oh et al., 2007; Oh et al., 2008) using heuristic algorithms to compute the cost of achieving individual parameters starting from the inputs, and search to approximate the optimal sequence of services that properly connect inputs to outputs. In addition, many of the AI planning approaches support the use of precondition and effects to describe services (Rao and Su, 2005). For instance, SWORD (Ponnekanti and Fox, 2002) is a developer toolkit for building composite web services using rule-based plan generation. In SWORD, a service is modeled by its pre-conditions and post-conditions and a web service is represented in the form of a Horn rule that denotes that post-conditions are achieved if the preconditions are true. However, looking through many of AI planning approaches, the pre-conditions express the required input parameters and effects specify expected services outputs which could be useful only for reasoning during planning. It is clear that this representation of pre- condition and effects cannot express other limitations of services such as service usage constraints as we discussed earlier.

In addition, service composition can be modeled as an optimization problem (Aggarwal et al., 2004; Channa et al., 2005). The optimization approach has appeared under different names such as QoS-driven or QoS-aware web service composition and web service composition optimization (Moghaddam and Davis, 2014). In (Hassine et al., 2006), Hassine et al. propose a constraint-based approach for the service composition problem. It provides a generic formalization of the web service composition problem as a constraint-optimization problem and then constructs a protocol to solve any composition problem by considering customer constraints. However, service constraints are not addressed in this approach.

1.2.2 Failure Recovery Approaches

The problem of failure recovery in web service composition is a problem that was discussed in many research works. This problem is addressed in different domains including *software adaptation* (Marconi and Pistore, 2009; Yan et al., 2010a; Laleh et al., 2014) and *web service composition transactions* (Gao et al., 2011; El Hadad et al., 2010; Dolog et al., 2014; Xu et al., 2016).

Software adaptation is a complementary domain that was devoted to the generation of mediators (also called software adapters, or simply adapters) to solve mismatch between components or services (Papazoglou et al., 2008; Yan et al., 2010a). *Replace-*

ment is one of the first adaptation approaches to react to a faulty service (Grigori et al., 2006; Cavallaro et al., 2009). Replacement is limited to 1-1 substitution and it focuses on finding a replacement for a broken service by another one. There are different solutions for this, such as finding a service that can use less input and produce more outputs than the original one. Replacement is an efficient solution in terms of computation time, however the limit of replacement is that a broken service often cannot be replaced by another unique service. *Re-composition* and *repair* are two approaches that support 1-n substitution. Re-composition re-builds the broken service by a 1-n substitution. Re-composition could also go further by using a completely different set of services and hence would correspond to an n-m substitution. Repair is also another solution that goes beyond the limits of service replacement while avoiding re-composition. This technique aims not only at keeping most of the above mentioned models as-is (i.e., not recompute them), but also takes benefit from them while computing a corrected composition. As such, repair is a form of heuristic and guided partial re-composition. In case of 1-1 substitution, repair performs as replacement and is as efficient. In other cases and for added needs, repair yields better computation time than re-composition while retrieving solutions of the same quality (Yan et al., 2010a).

Dynamic transaction support for web services is another approach to ensure that the composite service is executed correctly and achieves the overall desired result (Gao et al., 2011). Transactions are an approach employed to address system reliability and fault-tolerance (El Hadad et al., 2010) and the goal of service composition based on transactional properties is to ensures a reliable execution of the composite service. Traditional web services transaction processing mechanisms handle exception by forward and backward recovery approaches (El Hadad et al., 2010; Dolog et al., 2014). Backward recovery is essentially a form of rollback that unrolls the transaction and restores the original state of the system. Forward recovery approaches attempt to reach the original goal of the composite service by retrying or replacing components and continuing the process (Meyer et al., 2007; Yan et al., 2010b). In (Xu et al., 2016) a framework to optimize the success rate of transactional composite services is proposed. The framework considers the success rate of a service to include it as candidate in the composition process. In this way they improve the success rate of composite services completing successfully and thus reduce the need to employ failure recovery approaches.

1.3 Our Solution

Figure 2 shows the architecture of our solution including two main components: service composition, and composite service execution. The first component designs the composite services while taking into consideration user constraints and service constraints. The first step is to find all possible solutions for a composite service problem. Then, the constraints are adjusted inside the composite service to enable failure prediction, based on the algorithm we discuss in Section 2. Finally, a composite service package is assembled to create a composition including all solutions. The second component executes the composite service package considering the constraints of all individual services and recovers from failures.

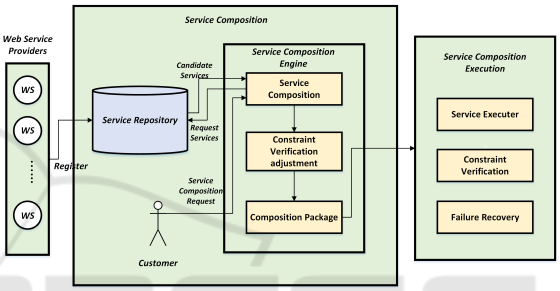

Figure 2: The architecture of our constraint-aware failure recovery approach.

2 COMPOSITE SERVICE MODEL

This section discusses our proposed approach for a constraint-aware composite service. First, we define basic concepts such as *Service* and *Constraint* that need to be used later in our model.

Definition 1. *A **Service** is defined as a tuple* $s =< I, O, C, E, QoS >$ *where:*

- ***I** is a set of ontology types representing the input parameters of the service.*
- ***O** is a set of ontology types representing the output parameters of the service.*
- ***C** is a set of constraint expressions representing limitations on services features.*
- ***E** is a set of ontology types representing parameters whose value are affected as a result of the execution of the service.*
- ***QoS** is the set of quality parameters of the service.*

In our definitions, we used ontology to define concepts (ontology type) and the relations between them. QoS criteria determine usability and utility of a service (Papazoglou, 2008).

We also need to define *Constraint* to specify the limitations on service features (input/output and QoS) that must be considered to ensure correct execution of services. A constraint is a function that maps a service feature to a set of values. To express constraints formally, we use the following definition:

Definition 2. *A **Constraint Expression** is an expression that can be evaluated to either true or false. For simplicity, we restrict ourselves to expressions of the form: $<feature><operator><literalValue>$, where:*

- $<feature>$ *represents a service feature.*
- $<operator>$ *represents operators such as* $=, \neq, <, >, \leq, \geq, \in, \subset, \supset, \subseteq, \supseteq$.
- $<literalValue>$ *represents a value or a set of values of the same data type as the expression feature.*

For example, $C = cost \geq 10$ expresses a constraint on the *cost* QoS feature of a service. In addition, there needs to be a mechanism to evaluate constraint satisfaction. Therefore, we define **Satisfaction Degree** as a mechanism to verify constraints.

Definition 3. *If c is a constraint expression and f is a service feature, **Satisfaction Degree** ($SD(f,c)$) is a function that calculates a quantitative measure to evaluate the satisfaction of the value assigned to f according to c.*

For example, if $C = cost \leq 10$, for any value assigned to cost ($cost \Leftarrow value$) :

$$SD(cost,c) = \begin{cases} true & value \leq 10 \\ false & \text{otherwise} \end{cases}$$

In addition, when there are more than one applicable constraint, **General Satisfaction Degree** verifies satisfactions of all constraints.

Definition 4. *If C is a set of constraints and f is a service feature, **General Satisfaction Degree** ($GSD(f,C)$) is a function that calculates a quantitative measure to evaluate the satisfaction of the value assigned to f according to all related constraints in C.*

If C is a set of constraints that includes n constraints that are applicable to f : $GSD(f,C) = \prod^n SD(f,c_i)$ In addition, if more than one constraint targets the same feature in a service, we define *intersection* as a mechanism to find all accepted values for the service feature according to both constraints.

Definition 5. *If f is a service feature, and c_1 and c_2 are two constraints on f, intersection ($Intersection(c_1,c_2)$) is a constraint that expresses all accepted values for f.*

For example, $c_1 = payment\_Method \notin \{Visa\}$ and $c_2 = payment\_Method \in \{Visa, MasterCard\}$ express two constraints on $payment\_Method$. Then $Intersection(c_1,c_2) = payment\_Method \in \{MasterCard\}$

Now we can define the service composition problem :

Definition 6. *A **Service Composition Request** R is a tuple $R =< I,O,QoS,C >$ where:*

- ***I** is the set of ontology types representing the input the customer can provide.*
- ***O** is the set of ontology types representing the output expected by the customer.*
- ***QoS** is the set of quality parameters expected from the service by the customer.*
- ***C** is the set of constraints representing limitations of customer-required features.*

The result of the service composition algorithm is the set of all plans that could accomplish the task expressed by the service composition request. Therefore, we define *plan* as:

Definition 7. *A **Plan** is a directed graph in which each node is a service that has sets of predecessor and successor services.*

The predecessor set represents the set of services that must be executed directly before the execution of the service node, and successors represent the set of services that are going to be executed directly after the execution of a service node in the plan. For example, for w_7 in *composition 3* (Figure 1), the predecessor and successor sets are: $predecessors(w_7) = \{w_5, w_6\}$ and $successors(w_7) = \emptyset$.

2.1 Constraint-aware Service Composition

In this section we discuss our solution to introduce constraint awareness in composite service. Our approach aims at modeling composite services such that their constraints can be verified at runtime to predict service failures and minimize service rollbacks. The planning search graph constructs a composition plan by Algorithm 1, which starts by initiating a list of temporary plans ($tempPlansList$) to produce input parameters (I)(Line 2). Then, it searches into SR (set of all available services) for services whose input parameters are available in the input request by the customer. This process is repeated and each time new services are selected form SR to generate new plans by extending already generated plans (Line 5-8). To extend a plan using a service, all input parameters of the service must have been produced by the plan (Line

6). Algorithm 2 adds the new service into the plan. If the new generated plan could satisfy design-time constraints such as the overall cost of the plan, it will be added to *newPlansList* (Line 10-11). If a plan produces the required outputs of the composition problem (O), it will be added to the list of successful plans (*goalPlansList*) (Line 12-16). The algorithm extends each newly generated plan and this process ends when the graph reaches a goal in which all parameters in O are produced or no more services can be added to the generated plans. If the output parameters cannot be produced in the search graph, the problem can not be solved.

Algorithm 1: Service Composition.

Input: R (composition request), SR (set of available services)
Output: list of plans or failure

```
1: tempPlansList.Add(initPlan)
2: newPlansList = Null
3: repeat
4:   for each plan in tempPlansList do
5:     for each service in SR do
6:       if (INPUT(service) ⊆ OUTPUT(plan)), and (service not in plan) then
7:         create Plan p
8:         p = addService(plan, service)
9:         if (CheckConstraints(p, R.C)) then
10:          newPlanList.Add(p)
11:          if (R.O ⊂ OUTPUT(p)) then
12:            p.Add(p_g)
13:            goalPlansList.Add(p)
14:            newPlanList.Remove(p)
15:          end if
16:        end if
17:      end if
18:    end for
19:  end for
20:  tempPlansList = newPlansList
21: until (newPlansList ≠ Null)
22: return goalPlansList
```

In our approach, services can be composed in sequence or in parallel. *AddService* (Algorithm 2) decides the order of the service in the newly created plan by specifying the predecessor services of each service in the plan. Therefore, for each input parameters of *newService*, Algorithm 2 searches back to find the latest service in the plan which produces the input parameter. Then, the service will be added to the set of predecessor services of *newService* (Line 4-5). Besides, the verification points of service constraints will be moved in the plan as early as possible toward the beginning of the plan in order to enable better failure prediction. Finally, Algorithm 2 calculates

Algorithm 2: Add Service.

Input: *plan* (composition plan), *newService* (A service)
Output: *plan* (composition plan)

```
1: I = INPUT(newService)
2: i = NumberOfService(plan)
3: while I ≠ ∅ do
4:   if (OUTPUT(plan.Service(i)) ∩ INPUT(newService) ≠ ∅) then
5:     newService.predessesor.Add(plan.Service(i))
6:     I = I − (OUTPUT(plan.Service(i)) ∩ INPUT(newService))
7:     for (each e ∈ (newService.C.features ∩ plan.Service(i).E)) do
8:       plan.Service(i).predecessors.constraints.Add(newService.Constraint(e))
9:     end for
10:  else
11:    break
12:  end if
13:  i = i − 1
14: end while
15: for (each preService ∈ (newService.predecessors)) do
16:   resp = calculateResponse(preService)
17:   if (plan.QoS.ResponseTime < resp + newService.QoS.ResponseTime) then
18:     plan.QoS.ResponseTime = newService.QoS.ResponseTime + resp
19:   end if
20: end for
21: plan.QoS.Cost = s.QoS.Cost + plan.QoS.Cost
22: plan.Add(newService)
23: return plan
```

the QoS criterion of the plan using the calculation approach discussed in (Lee et al., 1999; Li et al., 2016). In this approach, based on the order that concrete services are composed, the algorithm calculates the value of each QoS criterion of the plan.

2.2 Composite Service Package

The result of Algorithm 2 is a set of constraint-aware composite plans that could satisfy the initial service request by the user. During execution, the plan with the best utility function result value is selected. In Section 2.1, we discussed how our constraint verification approach aims at minimizing service usage rollbacks resulting from service failure and recovery.

Different recovery approaches that can be used in this situation. One approach can be replacement in which the execution system stops executing the failed plan and uses an alternative plan in the set of composite solutions. However, this solution might result in many rollbacks being required. Besides, like the situation discussed in Section 1.1 for executing w_1 and w_2 in *composition 1* and *composition 2* , the same services might be executed and their execution results need to be rolled back several times. Another solution can be using forward and backward failure recovery mechanisms to recover constraint-aware plans. However, all forward and backward recovery approaches (Dolog et al., 2014; Xu et al., 2016) require to replace, add or remove new services to the broken plan, while adding a new service could result in a need to repeat the constraint adjustment process discussed in Section 2.1.

As a result, we propose the notion of **composite service package** to manage failure recovery in a way to save waste executions and rollbacks, and does not impose constraint adjustment every time the composition plan changes.

Definition 8. *A **Composite Service Package** is a constraint-aware composition plan including all possible plans that can accomplish the same task.*

Figure 3 depicts a composition package that includes all composition plans discussed in Section 1.1. To create a composition package an algorithm is developed to integrate all possible composition plans into a composite package. We use the following operators discussed in (Hamadi and Benatallah, 2003; Wu et al., 2016) to describe a service composition work flow and then make a composition package.

- $\rightarrow$: Is an operator representing that the second service is executed when the execution of the first service is finished.
- $\oplus$: Is an operator representing that the two services are executed simultaneously.
- $\otimes$: Is an operator representing that one of the two services is selected to be executed.

First of all we add a service (w_0) to all the plans to make all the plans to have the same starting service. $w_0 =< I, O, C, E, QoS >$ is a service where $I = \emptyset$ and $O = I_{SC}$ where I_{SC} is the set of input parameters of the service composition problem. Now, to combine all possible composition plans in a composite package, we start with a plan with the highest utility function and then gradually add other plans to build the composition package. During execution of the package, the plans are going to be executed in this order in the composition package. In Algorithm 3, To make the composite service package, all plans (like $p = w_1 \rightarrow w_2... \rightarrow w_n$) need to be converted in a format as they only have $\rightarrow$ operator and each w_i could be a combination of services which can be executed in parallel ($\oplus$) or individual ($\otimes$). For example, *composition* 3 can be depicted as $w_1 \rightarrow w_V \rightarrow w_7$ when $w_V = w_5 \otimes w_6$.

If p_i and p_j are two plans such that: $p_i = w_0 \rightarrow w_1... \rightarrow w_k \rightarrow w_{k+1} \rightarrow .... \rightarrow w_x$ $p_j = w_0 \rightarrow w_1... \rightarrow w_k \rightarrow w_{k+1} \rightarrow .... \rightarrow w_y$ and we have : $V_i = w_k \rightarrow .... \rightarrow w_x$,$V_j = w_k \rightarrow .... \rightarrow w_y$ Then, these two plans are combined in a plan p: $p = w_0 \rightarrow w_1... \rightarrow w_k \rightarrow (V_i \otimes V_j)$ Algorithm 3 gets a set of constraint-aware plans and creates a composite service package out of these plans. It starts with considering the first plan as the composite package. Then, in each step, it adds a new plan to the composite package. Every time a plan needs to be added to the composition package, the intersection of the plan with the composite package should be found (Line 3-12). Then, based on what we discussed, the two plans should be combined together (Line 13-16).

Algorithm 3: Composite Package Creation.

Input: P (set of constraint-aware plans)
Output: pkg_plan (composition package)

```
1:  pkg_plan = P.getPlan()
2:  P = P − pkg_plan
3:  for (each p ∈ P) do
4:      i = 0
5:      repeat
6:          for (each service ∈ pkg_plan.service(i)) do
7:              if (service ∉ p.service(i)) then
8:                  break
9:              end if
10:         end for
11:         i = i + 1
12:     until (i ≤ pkg_plan.length)
13:     l1 = partialPlan(i + 1, pkg_plan.length)
14:     l2 = partialPlan(i + 1, p.length))
15:     tempPlan = l1 ⊗ l2
16:     pkg_plan = pkg_plan.part(i) → tempPlan
17: end for
18: return pkg_plan
```

3 COMPOSITE SERVICE PACKAGE EXECUTION

Algorithm 4 proposes a solution to execute a composite service package. As it is depicted in Figure 3, the structure of a composite service package is different from the structure of a simple composite service plan. In the following, we define the required concepts and then discuss the composite service package execution

algorithm in detail.

In AI planning for AWSC, a web service alters the state of the composite service upon execution. When a composite service is being executed, the state of the composite service changes step by step by execution of each component service. The composite service execution ends when all component services have been executed and the output of the final service in the plan is returned as the result of the execution of the composite service.

Definition 9. *A **State** is the set of all ontology types representing all features of the component services in the plan, each of them being initially assigned NULL values.*

We also need to define the way that a service changes a state value by its execution and in which condition a service can be applicable to a state.

Definition 10. *A service $w =< I,O,QoS,C,E >$ is **applicable** to a state $S = \{< T_1 \mapsto v_1 > ... < T_n \mapsto v_n >\}$, (where $\{T_1,T_2,...,T_n\}$ is a set of ontology types representing all features in a composition plan, and $\{v_1,...v_n\}$ are literal values of the same respective types) denoted as $S \succ w$, if all constraints of the service would be satisfied based on the values assigned to the features in S.*

Additionally, if a service applies to a state, a state transition function is applied to change the state of the composite service execution.

Definition 11. *When service w is applicable to state S ($S \succ w$), a **Service Transition Function** (γ) is applied to change the state of service execution to S' : $S' = \gamma(S,w)$.*

It should be noted that if all services of a composite service are composed in sequential order like *composition* 1 in the motivation scenario, the goal state will be calculated as: $G = (\gamma(\gamma(\gamma(S_0,w_1),w_2),w_4)$. However, if services are composed in parallel order (like W_5 and W_6 in *composition* 3), the goal state of the composition will be calculated as: $G = (\gamma(\gamma(\gamma(S_0,w_1),w_5) \bigcup \gamma(\gamma(S_0,w_1),w_6),w_7)$. Since a composite service package has a different structure compared to a regular composite service plan, we provide a different execution mechanism for it. Figure 3 shows the constraint-aware composite package of the scenario discussed in Section 1.1. The general idea behind the composite service package execution is to execute all plans inside the package one by one. During the execution of each plan, if the verification of a service constraint fails, the execution system prunes all plans that are related to the failed constraint. Then the execution continues with one of the remaining plans. For example for the composition package of our discussed scenario (Figure 3), the execution starts with the first service (w_1) of *composition 1*. Before the execution of w_1 all constraint moved before w_1, including C_1,C_{31},C_{41},C_{71}, will be verified. Then, if the verification of any of them fails, the package will prune all plans related to the failed constraint. For example, consider the case where C_{31} fails the verification before execution of w_1. It means that, based on the delivery address of the shopped item, the item cannot be shipped using w_3. As a result, any plan that includes w_3 (e.g. *composition 1*) will be pruned from the composite service package. This process will continue until all plans are pruned or at least one plan successfully completes the execution. Algorithm 4 represents the recursive approach that is designed to execute a composite service package. The execution starts from the root and in each step, based on the item (services/ operator) in the root, the execution proceeds. To execute a service (or set of services) in the root, the algorithm verifies that the set of constraints of the service in advance. If the verification of all constraints succeeds, it executes the services (Lines 1-7). However, if the verification fails, the composite package should be pruned.

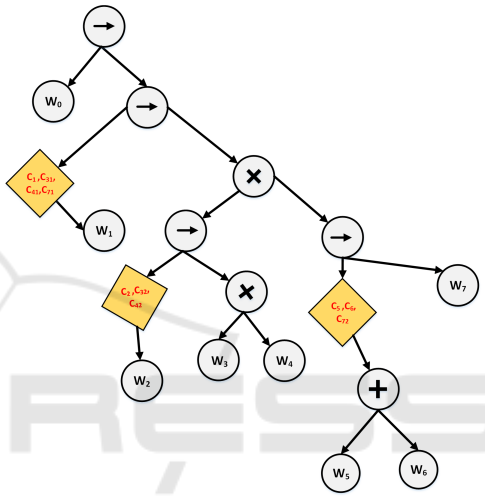

Figure 3: Composition Service Package with Constraints.

In addition, if there is an operator in the root, the algorithm should make the right decision to continue the execution of the plan (Lines 13-28). To do that, it first starts with computing the left and right sub-trees of the composition tree. Then, based on the operator in the root, it continues the execution. For $\rightarrow$, first the left sub-tree should be executed. Then, after it finished successfully, the right sub-tree will be executed. $\otimes$ means execution of one of the left or right sub-tree is enough. Finally, $\oplus$ means both sub-tree must be executed in parallel.

Algorithm 4: Composite Package Execution.

Input: pkg_plan (a composition package), S_0 (initial state of execution)
Output: either goal state or NULL

1: **if** $(IsService(pkg\_plan))$ **then**
2: **if** $(GSD(s_0, pkg\_plan.C))$ **then**
3: $StateList[pkg\_plan] = \gamma(pkg\_plan, S_0)$
4: **return** $StateList[pkg\_plan]$
5: **else**
6: $Prune(pkg\_plan)$
7: **end if**
8: **else**
9: $operator = getOperator(pkg\_plan)$
10: $t_1 = Left(pkg\_plan, operator)$
11: $t_2 = right(pkg\_plan, operator)$
12: **if** $(operator\ is\ \rightarrow)$ **then**
13: $temp = executionTree(t_1, S_0)$
14: $result = executionTree(t_2, temp)$
15: **end if**
16: **if** $(operator\ is\ \otimes)$ **then**
17: $result = Null$
18: $result = executionTree(t_1, S_0)$
19: **if** $(result\ is\ Null)$ **then**
20: $result = executionTree(t_2, S_0)$
21: **end if**
22: **end if**
23: **if** $(operator\ is\ \oplus)$ **then**
24: $temp1 = executionTree(t_1, S_0)$
25: $temp2 = executionTree(t_2, S_0)$
26: $result = combine(temp1, temp2)$
27: **end if**
28: **end if**
29: **return** $Null$

4 EXPERIMENTAL RESULTS

This section presents experimental results comparing the proposed composite service package execution approach with other failure recovery approaches including replacement, re-planning, re-composition and repair (Section 1.2). First we generated 5 different data sets using the WSC 2009 Testset Generator (WS-Challenge, 2009). Each data set contains a WSDL file which is the repository of web services. An OWL file lists the relationship between concepts and things. The number of services for each dataset are around 4000, and the number of concepts varies from 3000 to 3500 accordingly. In addition, the number of solutions in each dataset varies from 2 to 4 solutions. Since the generated data using this generator is not oriented to service composition considering constraints (C) and effects (E), in the following experiments we augmented the data sets with sets of effects to different services to meet our experimental needs. As we discussed in Section 2, E represents set of parameters whose value are affected as a result of the execution of the service. Therefore, for each service we consider the set of output parameters as the set E. In addition, in each service, for any item in E, a constraint is considered. However, instead of generating a constraint expression and a satisfaction degree for that, we defined a boolean variable to only consider the result of satisfaction of the constraint. This variable is initialized to true for all constraints. Then to simulate a service failure, we change values of all its constraints to false inside the plan.

To test the effectiveness of our approach we randomly failed services inside composite solutions and our composite service package. Then, different approaches were compared to see how many rollbacks were imposed as a result of the failure recovery. In re-composition, replacement and repair approaches, if the plan cannot be recovered, all the services until the broken point need to be rolled back. Each point is obtained from the average of 3 independent runs which in total is 15 different runs.

We compare all approaches from two aspects including the number of rollbacks (Figure 4) and the computation time (Figure 5). Figure 4 depicts the results of our experiments in terms of the number of rollbacks. It shows that re-planning imposes more rollbacks than other approaches. The reason is that every time a failure happens, re-planning needs to design the plan from the beginning. It is also clear that our approach imposes the fewest number of rollbacks compared to other approaches. This is due to the fact that our solution potentially reuses partially executed parts that are common between the current failed plan and its alternative selected after the failure. Our solution also allows to predict some failures that are going to happen later and to avoid going forward on a plan that we know is going to fail, thus saving rollbacks by *predicting* failure.

We also compared all approaches based on the computation time required to proceed with failure recovery (Figure 5), i.e. the time that the algorithm requires to do the recovery. Replacement is the fastest technique as it only requires one comparison with each available service in the repository. Re-planning has the worst time as it is the same as running the composition algorithm from the beginning after excluding failed services from the repository. In addition, the performance of our approach is not significantly different from repair and re-composition.

We also compared the success rate of the different approaches, i.e. the proportion of eventually suc-

cessful execution of a composite service. Among all approaches, re-planning and composite service packaging have better success rate as, unlike other approaches, they can step back from the broken point in the plan and start recovery. However, other approaches only move the plan forward during failure recovery. Replacement only looks for a service with the same same inputs/outputs and concepts. Repair and re-composition look for a new path to re-build the composition plan.

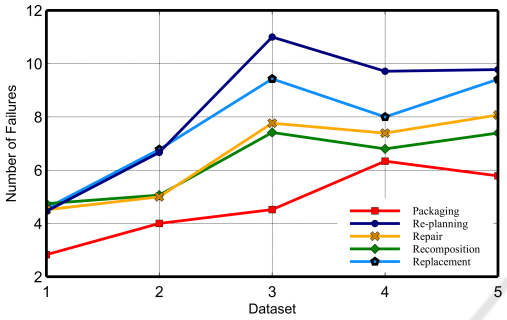

Figure 4: Number of rollbacks.

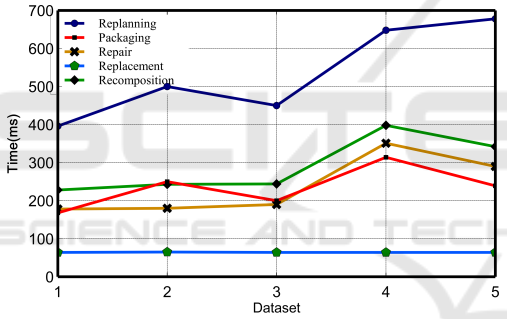

Figure 5: Time performance of different approaches.

5 CONCLUSION AND FUTURE WORK

In this paper, a constraint-aware failure recovery approach is proposed to first predict failures inside a composite service and, upon failure, proceed with recovery based on those predictions to reduce the number of rollbacks. We compared our approach to other failure recovery approaches from different perspectives. The experimental results demonstrate that our approach better minimizes the number rollbacks that are imposed as a result of failure recovery compared to other approaches. However, it is clear that our approach has limitations in cases where the number of possible solutions is potentially very large. We plan to use genetic algorithms to solve this problem as an optimization problem and come up with a local optimal set of solutions.

REFERENCES

Aggarwal, R., Verma, K., Miller, J., and Milnor, W. (2004). Constraint driven web service composition in meteors. In *Services Computing, 2004. (SCC 2004). Proceedings. 2004 IEEE International Conference on*, pages 23–30.

Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Mecella, M. (2003). Automatic composition of e-services that export their behavior. In *Service-Oriented Computing-ICSOC 2003*, pages 43–58. Springer.

Brogi, A. and Corfini, S. (2007). Behaviour-aware discovery of web service compositions. *International Journal of Web Services Research*, 4(3):1.

Cavallaro, L., Di Nitto, E., and Pradella, M. (2009). An automatic approach to enable replacement of conversational services. In *Service-Oriented Computing*, pages 159–174. Springer.

Channa, N., Li, S., Shaikh, A. W., and Fu, X. (2005). Constraint satisfaction in dynamic web service composition. In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 658–664. IEEE.

Dolog, P., Schäfer, M., and Nejdl, W. (2014). Design and management of web service transactions with forward recovery. In *Advanced Web Services*, pages 3–27. Springer.

El Hadad, J., Manouvrier, M., and Rukoz, M. (2010). Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing*, 3(1):73–85.

Gao, L., Urban, S. D., and Ramachandran, J. (2011). A survey of transactional issues for web service composition and recovery. *International Journal of Web and Grid Services*, 7(4):331–356.

Grigori, D., Corrales, J. C., and Bouzeghoub, M. (2006). Behavioral matchmaking for service retrieval. In *2006 IEEE International Conference on Web Services (ICWS'06)*, pages 145–152. IEEE.

Hamadi, R. and Benatallah, B. (2003). A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 191–200. Australian Computer Society, Inc.

Hashemian, S. V. and Mavaddat, F. (2005). A graph-based approach to web services composition. In *The 2005 Symposium on Applications and the Internet*, pages 183–189.

Hassine, A. B., Matsubara, S., and Ishida, T. (2006). A constraint-based approach to horizontal web service composition. In *The Semantic Web-ISWC 2006*, pages 130–143. Springer.

Laleh, T., Khodadadi, A., Mokhov, S. A., Paquet, J., and Yan, Y. (2014). Toward policy-based dynamic context-aware adaptation architecture for web service composition. In *Proceedings of C3S2E'14*, pages 158–163. Short paper.

Lécué, F. and Léger, A. (2006). A formal model for semantic web service composition. In *The Semantic Web-ISWC 2006*, pages 385–398. Springer.

Lee, C., Lehoezky, J., Rajkumar, R., and Siewiorek, D. (1999). On quality of service optimization with discrete qos options. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, pages 276–286.

Li, J., Yan, Y., and Lemire, D. (2016). Full solution indexing for top-k web service composition. *IEEE Transactions on Services Computing*, PP(99):1–1.

Liang, Q. A. and Su, S. Y. (2005). And/or graph and search algorithm for discovering composite web services. *International Journal of Web Services Research*, 2(4):48.

Marconi, A. and Pistore, M. (2009). Synthesis and composition of web services. In Bernardo, M., Padovani, L., and Zavattaro, G., editors, *Formal Methods for Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 89–157. Springer Berlin Heidelberg.

McIlraith, S. and Son, T. C. (2002). Adapting golog for composition of semantic web services. *KR*, 2:482–493.

Meyer, H., Kuropka, D., and Tröger, P. (2007). Asg-techniques of adaptivity. In *Autonomous and Adaptive Web Services*.

Moghaddam, M. and Davis, J. G. (2014). Service selection in web service composition: A comparative review of existing approaches. In *Web Services Foundations*, pages 321–346. Springer.

Oh, S.-C., Lee, D., and Kumara, S. R. (2008). Effective web service composition in diverse and large-scale service networks. *Services Computing, IEEE Transactions on*, 1(1):15–32.

Oh, S.-C., Lee, D., and Kumara, S. R. T. (2007). Web service planner (wspr): An effective and scalable web service composition algorithm. *Int. J. Web Service Res.*, 4(1):1–22.

Papazoglou, M. (2008). *Web services: principles and technology*. Pearson Education.

Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. (2008). Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255.

Ponnekanti, S. R. and Fox, A. (2002). Sword: A developer toolkit for web service composition. In *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, volume 45.

Rao, J. and Su, X. (2005). A survey of automated web service composition methods. In Cardoso, J. and Sheth, A., editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg.

Wang, P., Ding, Z., Jiang, C., and Zhou, M. (2014). Constraint-aware approach to web service composition. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(6):770–784.

Wang, P., Ding, Z., Jiang, C., Zhou, M., and Zheng, Y. (2015). Automatic web service composition based on uncertainty execution effects.

WS-Challenge (2009). Testsetgenerator2009. https://code.google.com/p/wsc-pku-tcs/downloads/list.

Wu, Q., Ishikawa, F., Zhu, Q., and Shin, D. H. (2016). Qos-aware multigranularity service composition: Modeling and optimization. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, PP(99):1–13.

Xu, J., Li, Z., Chi, H., Wang, M., Guan, C., Reiff-Marganiec, S., and Shen, H. (2016). Optimized composite service transactions through execution results prediction. In *Web Services (ICWS), 2016 IEEE International Conference on*, pages 690–693. IEEE.

Yan, Y., Poizat, P., and Zhao, L. (2010a). Repair vs. recomposition for broken service compositions. In *Service-Oriented Computing*, pages 152–166. Springer.

Yan, Y., Poizat, P., and Zhao, L. (2010b). Repairing service compositions in a changing world. In Lee, R., Ormandjieva, O., Abran, A., and Constantinides, C., editors, *Proceedings of SERA 2010 (selected papers)*, volume 296 of *Studies in Computational Intelligence*, pages 17–36. Springer Berlin Heidelberg.

Zheng, X. and Yan, Y. (2008). An efficient syntactic web service composition algorithm based on the planning graph model. In *Proceedings of the IEEE International Conference on Web Services (ICWS'08)*, pages 691–699. IEEE.