On Repairing Web Services Workflows

Thanh H. Nguyen(✉), Enrico Pontelli, and Tran Cao Son

New Mexico State University, Las Cruces, NM 88003, USA
{tnguyen,epontell,tson}@cs.nmsu.edu

Abstract. When a composite web service—i.e., a composition of individual web services—is executed and fails, it is desirable to reuse as much as possible the results that have been obtained thus far. For example, a travel agent, after receiving an order to arrange for a trip from *LA* to *NY* from a customer, would typically identify the flights and the hotels, obtain the confirmation from the customer, and place the reservations using the credit card information provided by the user; if something is wrong (e.g., at the last step, the credit card information was wrong), the travel agent would prefer to place the reservations using another means (e.g., a different card) instead of starting from the beginning.

This paper introduces an approach for dealing with service failures in the context of workflow execution. The paper defines the notion of a web service composition (WSC) problem and the notion of a *solution workflow* for a WSC problem. The paper describes two approaches to repair a partially executed workflow, with the goal of effectively reusing parts of the workflow that have been successfully executed. The usefulness of these approaches are demonstrated in an implementation using Answer Set Programming (ASP) in the well-known *shopping domain*.

Keywords: Repair · Reuse · Workflow · Web Services Composition

1 Introduction

The Semantics Web has been long considered as a killer application of the Internet that will, according to [1], *"unleash a revolution of new possibilities"* of the Web [1]. One of the key features of the Semantics Web is that it provides an environment suitable for intelligent agents to automatically: *(i)* discover and compose web services to create personalized services or workflows (i.e., *Web Services Composition (WSC)*); *(ii)* execute these personalized services whenever their users request; *(iii)* monitor such executions; and *(iv)* deal with failures of the services. These features are often provided by a WSC framework with two phases: one is responsible for the composition of web services and the other for the execution and monitoring the composition of web services.

Our interest, in this paper, is on the second phase of a WSC framework, *dealing with failures during the execution of a workflow*. This is because web services are inherently dynamic and cannot be expected to be stable all the time—developers often modify them, introduce faults, and modify APIs in unexpected manners. There are many different situations that can cause failures of web services [2,9]—ranging from *physical failures*, e.g., due to network failures, to *development failures*, e.g., due to incorrect APIs and incorrect logic, to *interaction faults*, e.g., due to incorrect parameter exchanges and misunderstood behavior. This paper focuses on the question of how to deal with physical failures.

There is a growing literature that addresses the problem of *recovery* when the execution of a web service fails. Several research contributions explore the problem of services monitoring, often based on *checkpointing* and oriented towards orchestration and choreography [13]. Proposed recovery methodologies include execution rollback to previous checkpoints and the use of redundancy to assist with server failures (e.g., [5,14,16]). Alternative approaches have explored the use of replacement of failed services in an attempt to repair a workflow (e.g., [4,15]). The idea of replacement has been expanded in [11], by allowing both rollback steps (with re-execution of failed services) as well as substitution of sequences of services with new workflows. A variety of studies have also proposed Several web service architectures that provide monitoring, fault detection, and exception event handlers have been described (e.g., [2,3,12]). Most of these approaches rely on static recovery techniques or relatively simple repetitions of the composition process. In [10], the authors propose a method based on partial-order planning that makes use of feedbacks from the plan execution to improve new plan search and to repair failed services. The method is illustrated using a shopping example. Unfortunately, the system available at http://sws.mcm.unisg.ch:8080/axis/services/MegashopService?wsdl is no longer active.

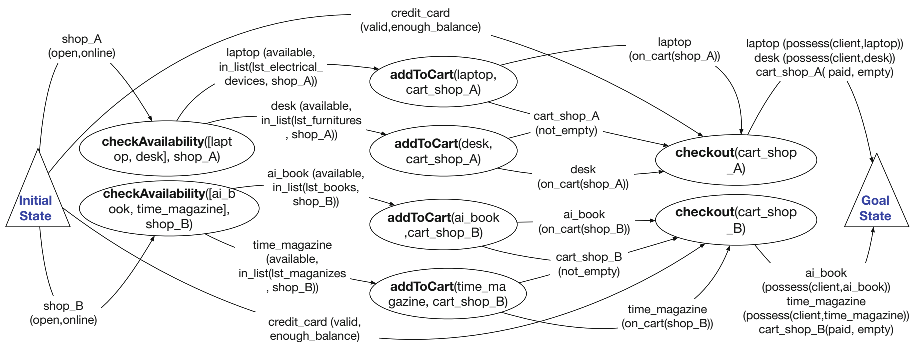

credit_card (valid,enough_balance)
shop_A (open,online)
laptop (available, in_list(lst_electrical_ devices, shop_A))
addToCart(laptop, cart_shop_A)
laptop (on_cart(shop_A))
laptop (possess(client,laptop)) desk (possess(client,desk)) cart_shop_A(paid, empty)
desk (available, in_list(lst_furnitures , shop_A))
cart_shop_A (not_empty)
checkAvailability([laptop, desk], shop_A)
addToCart(desk, cart_shop_A)
checkout(cart_shop_A)
desk (on_cart(shop_A))
ai_book (available, in_list(lst_books, shop_B))
Initial State
checkAvailability([ai_book, time_magazine], shop_B)
addToCart(ai_book ,cart_shop_B)
ai_book (on_cart(shop_B))
checkout(cart_shop_B)
Goal State
time_magazine (available, in_list(lst_maganizes , shop_B))
cart_shop_B (not_empty)
addToCart(time_magazine, cart_shop_B)
ai_book (possess(client,ai_book)) time_magazine (possess(client,time_magazine)) cart_shop_B(paid, empty)
shop_B (open,online)
credit_card (valid, enough_balance)
time_magazine (on_cart(shop_B))

Fig. 1. Shopping for some items from different web-sites (Web Shopping Domain)

In this paper, we investigate the problem of repairing a web services workflow whose execution failed. We develop a general framework for repairing web services workflows, that aims at *reusing as much as possible* the results obtained by an incomplete execution of the workflow. The framework could potentially be used in any WSC realization. We illustrate the framework by examples from the shopping domain.

2 Web Shopping Domain

Throughout the paper, we will illustrate our definitions using elements of the *Web shopping domain* as described in [10]. It consists of an ontology describing different classes of e-commerce shops (e.g., `electronic`, `furniture`, etc.), items sold in these shops (e.g., `laptops`, `books`, `desks`, etc.), containers (e.g., `shop_cart`), payment methods (e.g., `credit card payment`), and their subclasses (e.g., `shop_type_A` is a subclass of `shops`, `cart_shop_A` is a subclass of `shop_cart`, `ai_book` is a subclass of `books`, etc.). In addition, each type of shop provides a diversity of services and operations. They are listed below.

- *checkAvailability*: checks for the availability of a set of items. This service takes as input a list of items and returns the list of items and their metadata that are available in the shop. This service is available at all the shops.
- *addToCart*: places an item into a shopping cart. This service takes two inputs: a shopping cart and an item in the shop; the service produces an output which is the cart containing the added item. This service is provided in shops of type *A* and *B*.
- *removeFromCart*: removes an item from a shopping cart. It is the reverse of *addToCart* and is available at the shops of type *A* and *B*.
- *checkout*: purchases the items currently in the shopping cart; the service is available at shops of type *A* and *B*. It requires a valid credit card as an input. The output is that this cart is paid and the items belong to the client.
- *getItemsList*: returns the list of items that are in the shop's catalog. The input of this service is a shop and the output is the list of items in the shop's categories. This service is provided by shops of type *A* and *B*.
- *buytItem*: purchases a single item in shop. This service takes an item in the shop and a valid credit card as inputs. This service belongs to shops of type *A* and *C*.
- *getList*: works only for shops of type *C* and retrieves a list of all available items in the shop.

In shops of type *B*, every item needs to be placed into a shopping cart before it can be purchased (by the service *checkout*). In shops of type *C*, a selected item can be purchased directly by the service *buyItem*. Both purchasing methods (shopping cart or direct) can be used in shops of type *A*.

Figure 1 shows an example of a workflow over a WSC problem in the *web shopping domain*, aimed at purchasing some items (a `laptop`, an `ai_book`, a `time_magazine` and a `desk`) from different online stores with `credit_card` information that has been submitted by users. The initial state supplies a `credit_card` (and its information) to the services `checkout` or `buyItem`. The *Web Shopping Ontology* provides the information and status of the shops (`shop_A`, `shop_B` *etc.*) in the initial state, which are used in service `checkAvailability` to verify the availability of the requested item in shops as well as to retrieve the metadata of the requested items. Let us assume that, in this example, `credit_card` is `valid` and has `enough_balance` to use; these shops are `online` and `open`. In the goal state, `possess(client,X)` denotes that item *X* is owned by the *client*.

3 The Web Service Composition Problem

In order to formalize the notion of repair, we need a precise definition of the WSC problem and its solutions. The well-known abstract view of a WSC problem as a planning problem is easy to understand but assumes groundedness (i.e., actions are propositional terms)—which is not the case in our context, as parameters of a web service are often specified by *types* and will be instantiated only when the service is executed. We therefore start by defining the notion of a WSC problem.

An *abstract* resource is specified by its *type*, typically described as a class in an ontology. A *concrete* resource is specified by its *type* and *concrete data*, which is an instance of the resource type. A *named abstract/concrete resource* is a resource associated with a unique identifier. Let us denote with $\mathscr{T}$ the set of resource types. We use the notations (n,t,nil) and (n,t,d), where n is the name of the resource, $t \in \mathscr{T}$ is a resource type and d is an instance of t, to describe a *named abstract* and *named concrete* resource, respectively; *nil* represents an unknown value.

A Web service receives a set of named resources and produces a set of named resources. For example, the `getItemsList` service receives a shop object (e.g., `shop_A` of the type `shops`) and produces a list of all available items that are in the shop's catalogs (e.g., `list_catalog_A` of the type `list`). At the *specification level*, a Web service over a set of abstract resource types $\mathscr{T}$ is a pair $(a, e(a))$, where a is the service name and $e(a)$ is a tuple of pairs of *named abstract resources*, i.e., each element in $e(a)$ is of the form (in, out) where in, out denote sets of named resources. Each element in $e(a)$ is called a *precondition-effect* pair for a. For example, $e(\texttt{addToCart}) = (in_1, out_1)$ where $in_1 = \{(\texttt{cart\_id}, \texttt{shop\_cart}, \texttt{nil}), (\texttt{it\_id}, \texttt{item}, \texttt{nil})\}$ and $out_1 = \{(\texttt{cart\_id}, \texttt{shop\_cart}, \texttt{nil}), (\texttt{it\_id}, \texttt{item}, \texttt{nil})\}$.

The execution of a service a will take concrete data conforming to the specification in in and output concrete data of the type specified by out. For example, in Fig. 1, an instance of the `addToCart` service receives a shop cart object of the type `cart_shop_A`, a subclass of the type `shop_cart`, and an object of the type `laptop`, a subclass of the type `item`, as inputs, and produces as output a shopping cart containing the item.

Definition 1. *A Web service composition (WSC) problem $\mathscr{P}$ is a tuple $(\mathscr{T}, A, S_0, S_g)$ where*

- *$\mathscr{T}$ is a set of abstract resource types;*
- *A is a set of web services over $\mathscr{T}$;*
- *S_0 and S_g are two sets of concrete resources.*

Let $\mathscr{P} = (\mathscr{T}, A, S_0, S_g)$ be a web service composition problem. A *state s* of $\mathscr{P}$ is a set of concrete resources over $\mathscr{T}$. Let x be a set of abstract resources. We say that a set x_c of concrete resources is an instance of x if there exists a bijection b from x to x_c such that $b((n,t,nil)) = (n,t,d)$ for each $(n,t,nil) \in x$. Given a state s and a set of abstract resources x, we denote:

$$s|_x = \{(n,t,d) \mid (n,t,d) \in s, (n,t,nil) \in x\}.$$

We say that s contains an instance of x iff $s|_x$ is an instance of x. Given a state s and a service $a \in A$, the execution of a in s results in one of the three situations: **(i)** There exists a precondition-effect pair (i,o) of a such that $s|_i$ is an instance of i. In this case, we say that (i,o) is an active precondition-effect of a in s and the execution of a will produce an instance, denoted by $res(a,s)$, of o; **(ii)** There exists no precondition-effect pair (i,o) of a such that $s|_i$ is an instance of i. In this case, the execution of a will produce $\emptyset$, which will also be denoted by $res(a,s)$; or **(iii)** the execution of a fails, which will be denoted by $\perp$. In the following, a service $a \in A$ is *executable* in a state s if the cases (**i**) or (**ii**) occur. The WSC problem related to the example in Fig. 1 can be specified by $\mathscr{P}_s = (\mathscr{T}_s, A_s, S_0, S_g)$ where:

- $\mathscr{T}_s$ consists of the types (classes) in the ontology of the Web Shopping Domain;
- A_s consists of the services described in the previous section;
- $S_0 = \{$`credit_card(valid, enough_balance)`, `shop_A(open, online)`, ...$\}$
- $S_g = \{$`possess(client, laptop)`, `possess(client, desk)`, `possess(client, ai_book)`, `possess(client, time_magazine)`$\}$.

Definition 2. *A* workflow *over a WSC problem $\mathscr{P} = (\mathscr{T}, A, S_0, S_g)$ is a tuple $G = (V, E, v_0, v_g)$ where (V,E) is an acyclic directed graph with the set of nodes V and the set of labeled edges E, $v_0, v_g \in V$ are referred to as the* initial *and* goal state *of G, respectively, and*

- *each $v \in V \setminus \{v_0, v_g\}$ is associated to an action $a \in A$, denoted by $act(v)$;*
- *each $(u,v) \in E$ is labeled with a set of abstract resources, denoted by $l_E(u,v)$; and*
- *$\{x \mid (x,v_0) \in E\} = \emptyset$ and $\{x \mid (v_g,x) \in E\} = \emptyset$.*

A workflow over a WSC problem in the shopping domain is given Fig. 1. The two triangles represent the initial and goal state (v_0, v_g) respectively. Ellipses represent nodes of the graph, each node is associated to a service. Ingoing and outgoing links represent preconditions and effects of the service. For example, the top-left node is associated to `checkAvailability`, which requires a shop (in this case, `shop_A` from the initial state) and a set of items (that the client wishes to buy).

Given a workflow G over a problem $\mathscr{P}$, the execution of G starts from its initial state by sending concrete resources to its neighbors in accordance to the specification on the edges. Whenever all concrete resources from the predecessors of a node v are delivered to v, the service attached to v, $act(v)$, will be executed. If the execution is successful, i.e., it produces the proper concrete resources to be sent to the neighbors, then the execution continues; otherwise the execution of the workflow fails. The process continues until every service in the workflow is executed. The execution is said to be successful if the concrete resources specified at the goal state of G are produced. Formally, this process can be defined via a state function as follows.

Definition 3. *Let $G = (V, E, v_0, v_g)$ be a workflow over $\mathscr{P} = (\mathscr{T}, A, S_0, S_g)$. The* state function *of G, denoted by st, is a function that maps each node of G into a state of $\mathscr{P}$ or nil and is defined as follows.*

- *$st(v_0) = S_0$;*
- *for each $v \in V \setminus \{v_0\}$*
  - *(a) if there exists some $u \in V$ such that $(u,v) \in E$ and $st(u) = nil$ then $st(v) = nil$;*
  - *(b) otherwise, let $in(v) = \bigcup_{(u,v)\in E} st(u)|_{l_E(u,v)}$,*
    - *(b.1) if $v = v_g$ then $st(v) = in(v)$;*
    - *(b.2) if $v \neq v_g$ and $in(v) \cup res(act(v), in(v))$ is not an instance of $l_E(v,z)$ for some $(v,z) \in E$ then $st(v) = nil$;*

(b.3) if $v \neq v_g$ and Case (b.2) does not occur then $st(v) = in(v) \cup res(act(v), in(v))$.

We say that the execution of G succeeds if $st(v) \neq nil$ for every $v \in V$. Otherwise, the execution of G fails. *G is a* solution *of $\mathscr{P}$ if the execution of G succeeds and $S_g \subseteq st(v_g)$. Otherwise, G is not a solution of $\mathscr{P}$.*

In (Case b) of Definition 3, $in(v)$ is the set of concrete resources received by node v. $st(v)$ denotes the set of concrete resources which includes $in(v)$ and the result of the execution of $act(v)$ in $in(v)$. The situation $st(v) = nil$ indicates that the execution of the workflow at node v fails. Such situation can occur in different ways: **(i)** the execution of one of the predecessor of v failed (Case *a*); **(ii)** the execution of $act(v)$ does not result in proper concrete resources for the continuation of the execution of the workflow (Case *b.2*).

Observe that Definition 3 only considers G to be a solution of $\mathscr{P}$ if all services associated to G are executed successfully. This implies that G does not contain any redundant nodes, producing concrete resources not needed by any of its successors. This might sound too strong but it is reasonable for two reasons. First, the generation of G—similar to the generation of a plan—does not usually generate redundant nodes. Second, the definition could easily be relaxed to accommodate workflows with redundant nodes.

Observe also that Definition 3 assumes that communication between services is perfect and all services are executed. During the execution of a workflow, failures can happen when a service becomes unavailable. This could also be classified as a service failure. The execution monitoring server is responsible for dealing with this type of failures, as discussed in the next section.

4 Repair

Let $\mathscr{P} = (\mathscr{T}, A, S_0, S_g)$ be a WSC problem. Assume that $G = (V, E, v_0, v_g)$ is a workflow over $\mathscr{P}$. Furthermore, assume that G is a solution of $\mathscr{P}$ under the normal condition (e.g., communication between services is perfect, no machine failures, etc.), i.e., if the execution of G is successful then G will be a solution of $\mathscr{P}$. We are interested in situations where the execution of G is not successful due to the unavailability of a service attached to some node in G. In such a case, recovery measures are needed in order to achieve the goal of $\mathscr{P}$. For example, if the execution of G fails at node v, which is associated to the service $act(v)$, then a simple repair could consist of replacing $act(v)$ with another service $a \in A$ that takes the concrete resources at v and produces the concrete resources needed for the continuation of the execution of G. It is easy to see that this may not be always possible, due to the fact that no such service may exist. We call the process of identifying a new workflow G' that is a solution of $\mathscr{P}$, under the condition that the execution of G fails, as the *repair process*.

4.1 Formalization

Let $\mathscr{P} = (\mathscr{T}, A, S_0, S_g)$ be a WSC problem and $G = (V, E, v_0, v_g)$ be a workflow over $\mathscr{P}$. Let us assume that st is the state function of G.

Definition 4. *Let G be a workflow over $\mathscr{P}$, we define:*

$$A^{st}_{failed} = \{act(v) \mid st(v) = nil, st(u) \neq nil \text{ for all } u \text{ such that } (u,v) \in E\}.$$

We are interested in identifying another workflow G' which achieves the same goal and such that G' *reuses as much as possible the services in G that have been successfully executed.* It is easy to see that G' must not consider the services that are not available. Intuitively, a service $act(v)$ associated to a node v fails with respect to st if it does not allow the execution of the workflow to continue.

By G_{st} we denote the subgraph (V_{st}, E_{st}) of (V,E) such that $V_{st} = \{v \mid v \in V, st(v) \neq nil\}$ and $E_{st} = \{(v,u) \mid (v,u) \in E,\ v,u \in V_{st}\}$. For each $(u,v) \in E_{st}$, $l_{E_{st}}(u,v) = l_E(u,v)$. Thus, (V_{st}, E_{st}) is the graph containing all nodes whose services have been successfully executed. We explore two approaches for the repair process.

Planning from Failed State. The first alternative is to consider a new WSC problem whose initial state corresponds to the set of available concrete resources in G_{st}.

Let $V^{st}_0 = \bigcup_{v \in V_{st}} st(v)$. Intuitively, $V_0(st)$ denotes the set of concrete resources that are available. Let $\mathscr{P}' = (\mathscr{T}, A \setminus A^{st}_{failed}, V^{st}_0, S_g)$. We can achieve the goal of $\mathscr{P}$ by identifying a new workflow $G' = (V', E', v'_0, v'_g)$ over $\mathscr{P}'$. To evaluate how much G' reuses the executed services in G, we define the notion of a *reusable resource* as follows. Given a node v in a graph G, let us denote with $pre_G(v)$ the set of all predecessors of v in G—i.e., $u \in pre_G(v)$ iff there exists a non-empty path in G from u to v.

Definition 5. *Let $\mathscr{P}' = (\mathscr{T}, A \setminus A^{st}_{failed}, V^{st}_0, S_g)$ and $G' = (V', E', v'_0, v'_g)$ be a workflow over $\mathscr{P}'$ that is a solution of $\mathscr{P}'$. A concrete resource $(n,t,d) \in V^{st}_0 \setminus st(v_0)$ is said to be* reused *by G' if:*

- *for every $u \in pre_{G'}(v)$, $(n,t,d) \in st(u)$ and $(n,t,d) \notin res(act(u), in(u))$; and*
- *$(n,t,d) \in in(v)$ and, if (i,o) is the active precondition-effect of $act(v)$ in $in(v)$, then $(n,t,nil) \in i$.*

Intuitively, Definition 5 says that the concrete resource is reused if it is generated by G_{st} and is needed in G'. The above definition allows us to define the score of reusable resources as follows.

Definition 6. *Let G' be a solution of $\mathscr{P}'$. The amount of reusable resources of G' is denoted by $reused(G') = |\{(n,t,d) \mid (n,t,d) \in V^{st}_0 \setminus st(v_0), (n,t,d) \text{ is reused by } G'\}|$.*

We say that G' reuses more than G'', denoted by $G'' \prec_r G'$, if $reused(G') \geq reused(G'')$. It is easy to see that $\prec_r$ creates a transitive, reflexive, and antisymmetric relation among solutions of $\mathscr{P}'$. The identification of the workflow that reuses as much as possible of G is then equivalent to determining the solutions of $\mathscr{P}'$ which are maximal elements of $\prec_r$. The Implementation section will discuss how to compute such solutions.

Replanning with Successful Services. Replanning from the failed state might be useful. However, this will mean that we have to ignore the workflow G completely. In many situations, it is better to keep G as the original workflow might have components that have been included as desirable by the user. This idea also appears in the discussion of designing a workflow when users' preferences are taken into consideration [7]. For this reason, we develop an alternative approach to reuse the workflow G. This approach aims at keeping the services that have been executed successfully in the new workflow.

Let $G' = (V', E', v'_0, v'_g)$ be another solution of $\mathscr{P}$. We define a relation $\mapsto$ between G and G', as the minimal relation satisfying the following properties:

- $v_0 \mapsto v'_0$
- if $(x, y) \in E_{st}$, $(x', y') \in E'$, $x \mapsto x'$ and $act(y) = act(y')$ then $y \mapsto y'$

We next define the function $\pi : V' \longrightarrow \{0, 1\}$ as follows.

$$\pi(y') = \begin{cases} 1 & \text{if } \exists y \in V_{st}, y \mapsto y' \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Note that the second case in the definition of π occurs if either $y' = v'_0$ or if there is no node in G_{st} which is related to y'. Finally, we define a function: $score(G, G') = sum_{\{x' \in V'\}} \pi(x')$.

Definition 7. *A workflow G' is said to* re-use executed services and their relations as much as possible with current workflow G *if $score(G, G')$ is maximal.*

In Fig. 2, the blue arrow lines illustrate the relations $\mapsto$ between nodes in G and possible associated nodes in G' (e.g., *init* node in $G \mapsto$ *init* node in G', service node a in $G \mapsto$ service a in G', etc.); and G_{st} (grey area) is the part of G that was executed successfully.

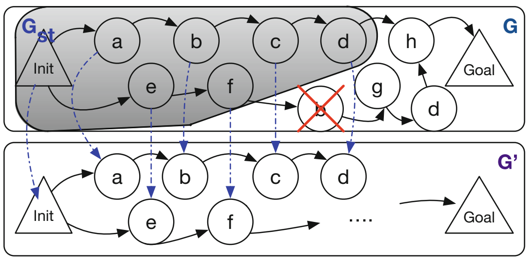


Fig. 2. Relation between G and G'

4.2 Implementation

In order to experiment with the notions of recovery discussed in the previous section, we obtained the source code from the authors of the code in the Phylotastic project as described in [8]. We implement the two notions of recovery on top of their implementation. The system described in the Phylotastic project has been developed using Answer Set Programming (ASP) and includes the *execution monitoring* module that captures the state of execution of a workflow. The implementation described in this paper will focus on the repairing phase. For reference, we denote with $\Pi(\mathscr{P})$ the planning module in the Phylotastic project and include the basic rules of the system below.

From Ontology to ASP Encoding: The ontology encoding the types, resources, services, etc. of the Web Shopping Domain is translated into an ASP program in $\Pi(\mathscr{P})$. This task is accomplished using a *translation program*. For example, a service in the class `pur_ol_op` takes `item` and `credit_card` as inputs and produces outputs `item`—has been purchased (`possess(client,item)`) and a `receipt`—will be sent to client by email (`have(receipt,sentByEmail)`). Operation `buyItem` is an instance of this class which purchases the available item if the provided `credit_card` is `valid` and `enough_balance`. The ASP encoding of this operation and its resources is the following:

Listing 1.1. ASP encoding for `buyItem` operation

```
class(purchase_op). class(pur_onl_op).
subclass(pur_onl_op,purchase_op).
operation(buyItem). type(buyItem,pur_onl_op).
class(credit_card). class(receipt).
class(payment_method). class(item).
subclass(credit_card,payment_method).
has_input(pur_onl_op,item_1,item).
has_input(pur_onl_op,card_1,credit_card).
has_output(pur_onl_op,item_1,item).
has_output(pur_onl_op,receipt_1,receipt).
input_spec(buyItem,item,item_1,have(item,available)).
input_spec(buyItem,credit_card,card_1,
    have(credit_card,valid)).
input_spec(buyItem,credit_card,card_1,
    have(credit_card,enough_balance)).
output_spec(buyItem,item,item_1,possess(client,item)).
output_spec(buyItem,receipt,receipt_1,
    have(receipt,sentByEmail)).
```

The predicate names are self explanatory.

Web Services Planning Engine: The planning engine of $\Pi(\mathscr{P})$ is similar to any planning engine implemented using ASP. It consists of different types of rules, divided into groups as follows.

– *Initial state:* The rule translates information given in S_0 to indicate that the data is available at time step 0. For example, `credit_card(valid,enough_balance)` is translated to

Listing 1.2. Initial State

```
init(credit_card,have(credit_card,valid)).
init(credit_card,have(credit_card,enough_balance)).
exists(X,F,0) :- init(X,F).
```

– *Planning:* In this listing, T denotes a step in the workflow; D_I/D_O the input and output type of a service, respectively; and $occ(A,T)$ says that A occurs at step T. Lines 1–2 enforce the precondition of a service. Lines 3–6 define *g_m* which indicates that an input I (type D_I) of A is provided by an output O (type D_O) at time $T_1 \leq T$. Line 7 defines $match/4$, which says that the input I of A is available at step T. Lines 8–10 generate action occurrences and make sure only actions whose preconditions are satisfied can be executed. Lines 11–15 define $map/8$ which maps between outputs produced at one step to inputs at later steps.

Listing 1.3. Planning Engine

```
1  {executable(A,T)} :- operation(A).
2  :- executable(A,T), input_spec(A,I,N,D),not match(A,I,D,T).
3  p_m(A,I,D_I,T,O,D_O,T_1):- operation(A),T_1≤T,
4      input_spec(A,I,N_I,D_I), exists(O,D_O,T_1),
5      subclass(O,I), subclass(D_O,D_I).
6  1{g_m(A,I,D_I,T,O,D_O,T_1):p_m(A,I,D_I,T,O,D_O,T_1)}1 :- step(T_1).
7  match(A,I,D_I,T) :- g_m(A,I,D_I,T,_,_,_).
8  1{occ(A,T) : operation(A)}1.
9  :- occ(A,T), not executable(A,T).
10 exists(O,D_O,T+1) :- occ(A,T), output_spec(A,O,N_O,D_O).
11 map(A,I,D_I,T,B,O,D_O,T_1):- occ(A,T),T>=T_1,
12     occ(B,T_1-1),g_m(A,I,D_I,T,O,D_O,T_1),
13     input_spec(A,I,N_I,D_I), output_spec(B,O,N_O,D_O).
14 map(A,I,D_I,T,initG,O,D_O,0):- occ(A,T),
15     g_m(A,I,D_I,T,O,D_O,0), input_spec(A,I,N_I,D_I).
```

The program $\Pi(\mathscr{P})$ will also contain a generic rule of the form "`:- not goal(n)`" where $goal(n)$ indicates that the goal is satisfied at step n. For a constant n, $\Pi(\mathscr{P},n)$ denotes the program $\Pi(\mathscr{P})$ with steps taken values in $\{1,\dots,n\}$ with the goal checking rule at n. Answer sets of $\Pi(\mathscr{P},n)$ represent workflows solving $\mathscr{P}$. The current execution and monitoring system of the Phylotastic project is responsible for the execution of workflows generated by $\Pi(\mathscr{P},n)$. It *stops* whenever a failure occurs. $\Pi(\mathscr{P},n)$ is enhanced as follows.

Repairing Method 1: Planning from Failed State. We encode the available resources at the failed state by ASP atoms of the form $res\_gen/4$. We alter the execution and monitoring system to record this information. To plan from the failed state, we only need to add the available resources to the initial state, remove the failed services by encoding them as $failed(op,\_)$ to prevent Π to consider these services in the planning phase. Let $F(\mathscr{P})$ denote the set of facts of the form $res\_gen/4$ or $failed/2$ that is supplied by the execution monitoring system when a failure occurs.

To compute the amount of reused resources, we add the rules[1] in Listing 1.4 to $\Pi(\mathscr{P})$. In Listing 1.4, the first rule (Lines 1–2) records the successful execution of operation X at step T and defines *res_prod*, the resource R named N has been produced by an operation X at step T. The lines 3–5 define the predicate *res_reuse*, which says that the resource R named N of the type D_R and data Data_R is reused. Line 6 counts the number of reused resources. Line 7 enables the identification of answer sets with maximal number of reused resources and Line 8 prevents reuse of failed services.

Listing 1.4. Planning From Failed State (Π_{pff})

```
1  res_prod(R,N,D_R):- res_gen(R,N,D_R,Data_R),
2       output_spec(X,R,N,D_R), occ(X,T).
3  res_reuse(R,N,D_R,Data_R) :- occ(X,T),
4       res_gen(R,N,D_R,Data_R), not res_prod(R,N,D_R),
5       map(X,I,D_I,T,initG,R,D_R,0).
6  reused(V) :- V = #count{R,N,D_R,Data_R : res_reuse(R,N,D_R,Data_R
       )}.
7  #maximize{V : reused(V)}.
8  :- failed(F,T), is_used_op(F).
```

Let $\Pi_1^f(n) = \Pi(\mathscr{P},n) \cup \Pi_{pff} \cup F(\mathscr{P})$. We can show the following:

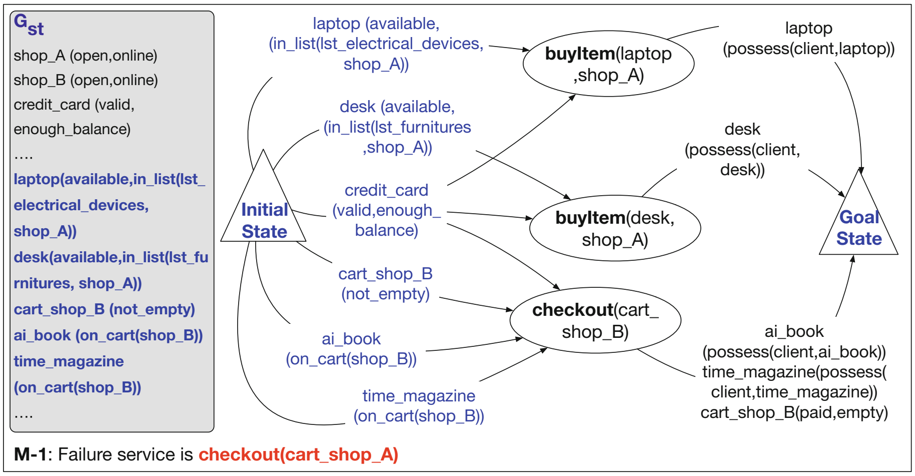
Proposition 1. *If A is an answer set of $\Pi_1^f(n)$ then A encodes a workflow solution of $\mathscr{P}$ that does not include any failed service and reuses the maximal number of resources specified in $F(\mathscr{P})$.*

Proof. The fact that A satisfies $\Pi(\mathscr{P},n)$ indicates that A encodes a workflow solution of $\mathscr{P}$. The rules on Lines 6–8 (Listing 1.4) ensure the other properties of the solution. $\square$

Repairing Method 2: Replanning with Successful Services. Let G_{st} be the workflow whose execution fails. We assume that G_{st} and the services that have been executed successfully are encoded by a program $F(G_{st})$, which consists of facts of the form *old_occ_exe*/2 or $old\_map\_exe(S,I,D_I,T_1,S_0,O,D_O,T_0)$. We add to $\Pi(\mathscr{P},n)$ a new set of rules Π_{rss} (Listing 1.5) and $F(G_{st})$ and generate new solution G' for $\mathscr{P}$ such that $score(G_{st},G')$ is maximal. In addition, $F(G_{st})$ also records failed services.

The program Π_{rss} (Listing 1.5) implements the function $\mapsto$. The first rule (Line 1) says that we will map the initial state of G_{st} to the initial state of G'. Other rules defined φ (Lines 2–6, 7–11) extend the $\mapsto$ relation whenever possible. In these rules, `s_equal(Y,Y')` represents the fact that two services Y and Y' are equivalent in terms of functionality. The rules defining π (Lines 12–14) compute the value of π as defined in Eq. 1. Π_{rss} computes the number of reused services, instructs the solver to find answer sets containing the maximal number of reused services (Lines 15–16), and makes sure that the generated workflow G' does not include failed services (Line 17–18).

[1] The rules have been simplified somewhat for readability.



G_st
shop_A (open,online)
shop_B (open,online)
credit_card (valid, enough_balance)
....
laptop(available,in_list(lst_electrical_devices, shop_A))
desk(available,in_list(lst_furnitures, shop_A))
cart_shop_B (not_empty)
ai_book (on_cart(shop_B))
time_magazine (on_cart(shop_B))
....
Initial State
laptop (available, (in_list(lst_electrical_devices, shop_A))
desk (available, (in_list(lst_furnitures ,shop_A))
credit_card (valid,enough_balance)
cart_shop_B (not_empty)
ai_book (on_cart(shop_B))
time_magazine (on_cart(shop_B))
buyItem(laptop ,shop_A)
buyItem(desk, shop_A)
checkout(cart_shop_B)
laptop (possess(client,laptop))
desk (possess(client, desk))
ai_book (possess(client,ai_book)) time_magazine(possess(client,time_magazine)) cart_shop_B(paid,empty)
Goal State
M-1: Failure service is checkout(cart_shop_A)

Fig. 3. Recovery workflow (Web Shopping Domain): Replanning from Failed State

Listing 1.5. Replanning with Successful Services (Π_{rss})

```
φ(initG,0,initG',0).
φ(Y,T_1,Y',T_3) :- φ(initG,0,initG',0),
    old_occ_exe(Y,T_1), occ(Y',T_3),
    old_map_exe(Y,I,DF_I,T_1,initG,O,DF_O,0),
    map(Y',I,DF_I,T_3,initG',O,DF_O,0),
    s_equal(Y,Y').
φ(Y,T_1,Y',T_3) :- φ(X,T_2,X',T_4),
    old_occ_exe(Y,T_1), occ(Y',T_3),
    old_map_exe(Y,I,DF_I,T_1,X,O,DF_O,T_2+1),
    map(Y',I,DF_I,T_3,X',O,DF_O,T_4+1),
    T_1>=T_2+1,T_3>=T_4+1,s_equal(Y,Y').
π(initG',0).
π(Y',0) :- occ(Y',T_3), not φ(_,_,Y',T_3).
π(Y',1) :- occ(Y',T_3), φ(_,_,Y',T_3).
score(V_φ) :- V_φ = #sum{V_Y',Y' : π(Y', V_Y')}.
#maximize{V_φ : score(V_φ)}.
:- failed(F,T_F), occ(F,T), not succ(F,T).
succ(F,T) :- failed(F,T_F), occ(Y,T), Y = F, φ(F,_,Y,T).
```

Proposition 2. *If A is an answer set of $\Pi_2^f(n) = \Pi(\mathscr{P},n) \cup \Pi_{rss} \cup F(G_{st})$ then A encodes a workflow solution of $\mathscr{P}$ that does not include any failed service and reuses the maximal number of services specified in $F(G_{st})$.*

Proof. Similar to Proposition 1. □

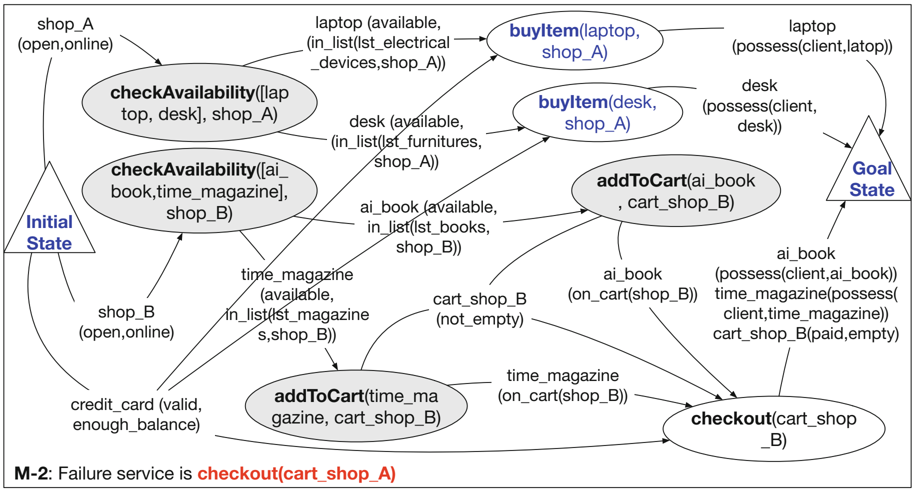

shop_A (open,online)
laptop (available, (in_list(lst_electrical _devices,shop_A))
buyItem(laptop, shop_A)
laptop (possess(client,latop))
checkAvailability([lap top, desk], shop_A)
buyItem(desk, shop_A)
desk (possess(client, desk))
desk (available, (in_list(lst_furnitures, shop_A))
Goal State
checkAvailability([ai_ book,time_magazine], shop_B)
addToCart(ai_book , cart_shop_B)
Initial State
ai_book (available, in_list(lst_books, shop_B))
ai_book (possess(client,ai_book)) time_magazine(possess(client,time_magazine)) cart_shop_B(paid,empty)
time_magazine (available, in_list(lst_magazine s,shop_B))
ai_book (on_cart(shop_B))
shop_B (open,online)
cart_shop_B (not_empty)
time_magazine (on_cart(shop_B))
addToCart(time_ma gazine, cart_shop_B)
credit_card (valid, enough_balance)
checkout(cart_shop _B)
M-2: Failure service is checkout(cart_shop_A)

Fig. 4. Recovery workflow (Web Shopping): Replanning with Successful Services (Color figure online)

4.3 Experimental Evaluation

Web Shopping Domain. We experimented the approaches with the problem described in the second section[2]. A failure of service (`checkout(cart_shop_A)`) is injected during the execution of the workflow generated in Fig. 1. Figure 3 shows a new workflow generated by $\Pi_1^f(3)$ that uses the planning from the failed state method (0.5 s). Figure 4 shows a new workflow generated by $\Pi_2^f(7)$ that uses the replanning with successful services method (2 s). Observe that the repaired workflow on the left, generated by $\Pi_1^f(3)$, utilizes the available resources and is simpler comparing to the original one. On the other hand, the workflow on the right, generated by $\Pi_2^f(7)$, contains new services that replace the failed service and is able to reuse the majority of the executed services.

In Fig. 3, the new initial state includes all *concrete resources* which have been produced successfully before failure point when *executing* the original workflow. For example, `ai_book(on_cart(shop_B))` is the concrete resource produced by service `addToCard` when executing the original workflow. Program $\Pi_1^f(3)$ generated a new workflow from new initial state to original goal state and reused concrete resource `ai_book(on_cart(shop_B))`. In Fig. 4, the gray background eclipse nodes and their links simulate whole or a part of executed structure G_{st} while repairing services are represented by blue text nodes.

Phylotastic Domain. We did further experimental evaluation to evaluate the two repairing methods with problems in the *Phylotastic* domain as well. The detail information about *Phylotastic* domain is described in [8]. Basically, the *Phylotastic Ontology* is

[2] We used a computer running Ubuntu 16.4 LTS, 8GB DDR3, 2.5 GHz Intel-Core i5, and ASP solver `clingo`.

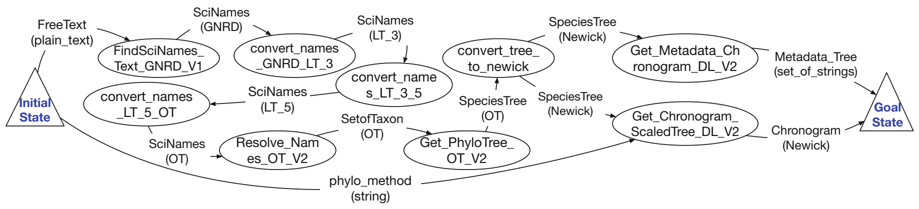


Fig. 5. Generating a chronogram and its metadata from free plain text

a services repository that deals with the manipulation of services (e.g., names, species, phyloreferences) and representations of evolutionary knowledge (e.g., taxonomies, phylogenies). There are some primary classes of services such as `names_operation`, `tree_operation`, `taxon_operation`, etc. The results of repairing in *Phylotastic* domain are discussed below.

– **Use case 1: From Free Plain Text to a Chronogram and its Metadata.** We experimented with a WSC problem in the *Phylotastic* domain, aimed at creating a `chronogram`, a scaled species phylogeny with branch lengths, in `newick` format, along with its metadata, from a plain text document. The initial state supplies a `FreeText` (more precisely, an object of the type `FreeText`) to the service `FindSciNames_Text_GRND_V1` (type of `names_extraction_text`) and a string (`phylo_method`) to the service `Get_Chronogram_ScaledTree_DL_V2` (type of `tree_transformation`), which also needs a `speciesTree` in `newick` format to produce the `chronogram`, etc. A workflow solving for this problem is given in Fig. 5. In this experiment, we consider two scenarios: (**S1**) The service `convert_tree_to_newick` fails; and (**S2**) the service `Get_Chronogram_ScaledTree_DL_V2` fails. The workflows generated using the two methods are depicted in Fig. 6. The two workflows on the left are generated from the failed state. Available resources are

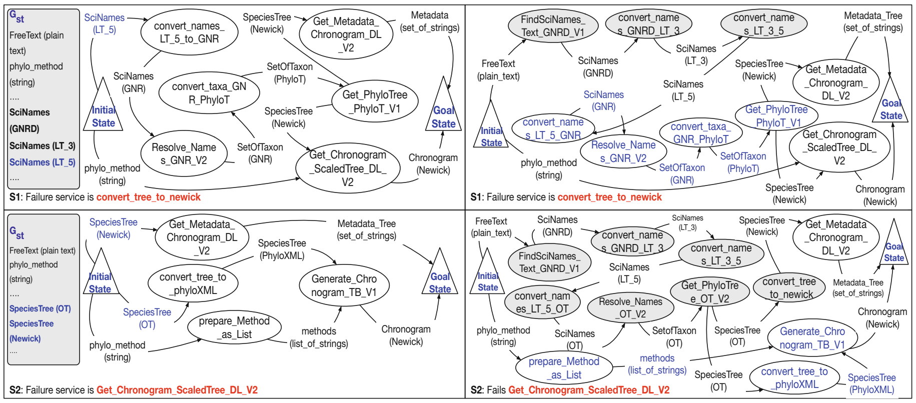


Fig. 6. Repaired workflows: *Phylotastic* generating a chronogram and its metadata (Color figure online)

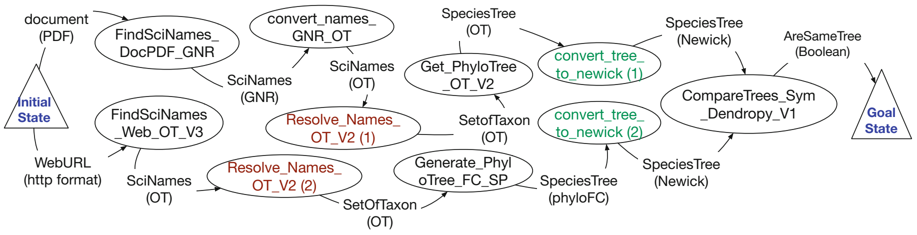


Fig. 7. Compare species trees from different sources

listed in the gray box. The reused resources are highlighted in Blue. In both scenarios, the new workflow differs significant from the original one (e.g., new services are used). The two workflows on the right are generated by replanning using successful services. The new services that need to be inserted are highlighted in Blue. As it can be seen, almost a half of all executed services are reused in (**S1**) and only a few are added in (**S2**).

– **Use case 2: Compare Species Trees from Different Sources.** In this use case, an user provides inputs data including a document (`PDF` format), and a Web address (`http_URL` format). Each document contains information about a phylogeny tree. The requirement is to examine whether or not the two phylogeny trees generated from these inputs sources (document and Web-page content) have the same phylogeny structure (Fig. 7). There are two concrete services that are used more than one time in the workflow: `Resolve_Names_OT_V2` and `convert_tree_to_newick`. They are drawn with a number next to it (1 and 2) in Figs. 7 and 8 in order to identify which service will be executed before another in the execution ordering[3]. For example, `Resolve_Names_OT_V2(1)` and `Resolve_Names_OT_V2(2)` describe the same service and `Resolve_Names_OT_V2(1)` is executed before `Resolve_Names_OT_V2(2)`. Again, we inject two different service failures in two scenarios: (**S1**) The failure is at the very end of the process (`CompareTrees_Sym_Dendropy_V1`); and (**S2**) The failure is at `convert_tree_to_newick(2)` (meaning that the first execution of the service `convert_tree_to_newick` succeed while the second execution fails). Figure 8 depicts the four recovery workflows for use case 2. The left two are workflows generated from the failed state while the right ones are generated by replanning with using successful services.

Comparison Between Two Methods. We close this section with a brief discussion on the advantages and disadvantages of the two methods for repairing failed workflows. Clearly, both aim at *reusing as much as possible the results obtained from the incomplete execution of the workflow* but with a slight different focus. Method 2 (*Replanning with Successful Services*) attempts to take advantage of the information in the original workflow and Method 1 (*Planning from Failed State*) ignores this information. This leads to the following main differences:

[3] The ordering is done so we can experiment with the failure of the services. It is also possible for the order to be reverse.

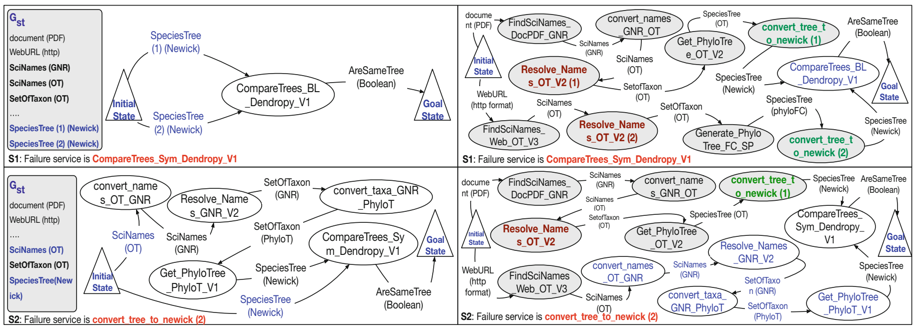


Fig. 8. Repaired workflows: *Phylotastic* compare species trees from different sources

– **Planning from Failed State.** In this method, the replanning process often starts from a state that is richer in resources than the initial state of the original problem. This is due to the fact that the execution of a service does not remove the outputs of services that have been successfully executed earlier. Given that the original workflow is minimal under certain metric (e.g., shortest plan) then the new initial state is closer to the goal than the original one. Therefore, Method 1 could be faster than Method 2—it can be observed in the experiment.
– **Replanning with Successful Services.** The replanning in this method is more complex than Method 1. It involves the generation of a new workflow (G') and then a *subgraph isomorphism* between the new workflow and the partial executed workflow (G' and G_{st}). In our implementation, this is done in a single ASP program. This is the added overhead and is the main reason for Method 2 to take longer to find a new plan as in the experiments. On the other hand, as suggested in [7], maintaining the skeleton of the original workflow is a desirable feature of the replanning process.

5 Conclusion, Discussion, and Future Work

We formally defined the notion of a WSC problem that enables the introduction of two methods for repairing a workflow whose execution fails when a services fails. The most interesting feature of the two new methods lies in the precise definition of the rather relative statement of *reusing as much as possible what has been executed*. In one method, the focus is on the available resources that have been produced by the executed services and a new workflow is generated from the failed state. In another method, the focus is on reusing the original workflow in generating the new one. We experimentally evaluate the proposed methods in the shopping domain by integrating them into an existing WSC framework. (Available online at: http://workflow.phylotastic.org)

We note that the WSC problem defined in this paper inherits several characteristics of a WSC problem defined in the literature (e.g., as in [6]), which views the WSC problem as a planning problem and its solution is generally a graph (a workflow) and not a sequential plan. On the other hand, due to fact that web services need resources to start their execution but do not usually *remove* them, the replanning problem with respect to a WSC problem is different from replanning in general. Our first method of replanning relies on this property of WSC problems. Last but not least, we are not aware of any comparable WSC system that we could use in our experiments. The several WSC systems reviewed in the introduction are no longer functional or inaccessible. Our intermediate goal is to identify potential applications that could benefit by the proposed methods and allows us to experiment and validate the scalability of the proposed system.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The semantics web. Sci. Am. **284**(5), 34–43 (2001)
2. Chan, K.S.M., Bishop, J., Steyn, J., Baresi, L., Guinea, S.: A fault taxonomy for web service composition. In: Di Nitto, E., Ripeanu, M. (eds.) ICSOC 2007. LNCS, vol. 4907, pp. 363–375. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-93851-4_36
3. Chen, H.P., Zhang, C.: A queueing-theory-based fault detection mechanism for SOA-based applications. In: The 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007), pp. 157–166 (2007)
4. Erradi, A., Maheshwari, P., Tosic, V.: Recovery policies for enhancing web services reliability. In: 2006 IEEE International Conference on Web Services (ICWS 2006), pp. 189–196 (2006)
5. Mansour, H.E., Dillon, T.S.: Dependability and rollback recovery for composite web services. IEEE Trans. Serv. Comput. **4**, 328–339 (2011)
6. McIlraith, S., Son, T., Zeng, H.: Semantic Web services. IEEE Intell. Syst. **16**(2), 46–53 (2001). (Special Issue on the Semantic Web)
7. Nguyen, T., Pontelli, E., Son, T.: Phylotastic: an experiment in creating, manipulating, and evolving phylogenetic biology workflows using logic programming. Theory Pract. Logic Program. **18**(3–4), 656–672 (2018)
8. Nguyen, T.H., Son, T.C., Pontelli, E.: Automatic web services composition for phylotastic. In: Calimeri, F., Hamlen, K., Leone, N. (eds.) PADL 2018. LNCS, vol. 10702, pp. 186–202. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73305-0_13
9. Nwana, H.: Software agents: an overview. Knowl. Eng. Rev. **11**(3), 205–244 (1996)
10. Peer, J.: A POP-based replanning agent for automatic web service composition. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 47–61. Springer, Heidelberg (2005). https://doi.org/10.1007/11431053_4
11. Saboohi, H., Amini, A., Abolhassani, H.: Failure recovery of composite semantic web services using subgraph replacement. In: 2008 International Conference on Computer and Communication Engineering, pp. 489–493 (2008)
12. Vaculín, R., Wiesner, K., Sycara, K.P.: Exception handling and recovery of semantic web services. In: Fourth International Conference on Networking and Services (ICNS 2008), pp. 217–222 (2008)
13. Vargas-Santiago, M., Hernández, S.E.P., Rosales, L.A.M., Kacem, H.H.: Survey on web services fault tolerance approaches based on checkpointing mechanisms. JSW **12**, 507–525 (2017)
14. Yin, J., Chen, H., Deng, S., Wu, Z., Pu, C.: A dependable ESB framework for service integration. IEEE Internet Comput. **13**, 26–34 (2009)
15. Yin, K., Zhou, B., Zhang, S., Xu, B., Chen, Y.: Qos-aware services replacement of web service composition. In: 2009 International Conference on Information Technology and Computer Science vol. 2, pp. 271–274 (2009)
16. Zhao, W.: Design and implementation of a Byzantine fault tolerance framework for web services. J. Syst. Softw. **82**, 1004–1015 (2009)