Repair vs. Recomposition for Broken Service Compositions

Yuhong Yan[1], Pascal Poizat[2,3], and Ludeng Zhao[1]

[1] Concordia University, Montreal, Canada
yuhong@encs.concordia.ca
[2] University of Evry Val d'Essonne, Evry, France
[3] LRI UMR 8623 CNRS, Orsay, France
pascal.poizat@lri.fr

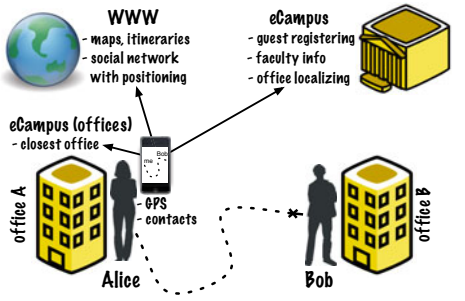
Abstract. Service composition supports the automatic construction of value-added distributed applications. However, this is nowadays mainly a static affair, with compositions being built once and for all. Moving from a static to a dynamic world, where both available services and needs may change, requires automated techniques to correct broken compositions. Recomposition is a working solution but it requires to rebuild composition models from scratch. With graph planning as the service composition framework, we propose repair as an alternative to recomposition. Rather than discarding broken compositions, repair reuses and corrects them for fast generating new service compositions. Our approach is completely tool-supported. This enables us to compare repair and recomposition using both a case study and a data set from a service composition benchmark framework.

1 Introduction and Motivating Example

Software architects benefit from automatic service composition (ASC) techniques and tools [27,8,16] to foster the rapid design, implementation and deployment of distributed applications. Still, ASC enables a more dynamic and on-demand way to develop software, where end-users directly expose their needs to composition engines. The conjunction of technical developments (ubiquitous computing and service pervasiveness), social usages (smart devices equipment rate and user nomadism), together with an adequate business model (cloud computing) makes this vision a close reality. Still, ASC has to support the automatic evolution of compositions, taking into account changes both in the needs of the (possibly mobile) end-users and in the services availability.

A motivating example: the eMeet scenario. Let us imagine the following scenario. Alice is on her way to meet a friend, Bob, who works at a University. To reach him, Alice needs an itinerary map. Since she could get out of her way inadvertently, Alice wants that itinerary can be updated. To achieve her goals, Alice can use services in her vicinity (Fig. 1): M:map (localized map), M:way (itinerary map from location p_1 to location p_2), F:gps (friend position),

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 152–166, 2010.



name	inputs	outputs
maps and itineraries service		
M:map	gps_p	map_p
M:way	gps_{p_1}, gps_{p_2}	$\mathsf{way}_{p_1 p_2}$
social network w/ positioning service		
F:gps	fid_{p_1}, fp_{p_1}, fid_{p_2}	gps_{p_2}
eCampus service		
U:reg	n_{p_1}, n_{p_2}, lab_{p_2}	vt_{p_1}
U:info	vt_{p_1}, n_{p_2}, lab_{p_2}	id_{p_2}, off_{p_2}
U:ogps	vt_{p_1}, off_p	gps_p
U:office	vt_{p_1}	off_{p_1}
on-person services (smart phone)		
_:gps	–	gps_a
_:info	n_b	fid_b, lab_b

$P{=}\{a, b\},\ p, p_1, p_2 \in P,\ p_1 \neq p_2$

Fig. 1. eMeet scenario – architecture (left) and services (right)

U:reg (guest registration), U:info (faculty member information), U:ogps (office location), U:office (closest office), _:gps (own position), _info (friend id and lab). Services are equipped with semantic descriptions to support their automatic discovery and composition. These refer to GPS positions (gps), social network ids and passwords (fid, fp), names and labs (n, lab), faculty ids, office ids, and visitor authorization tokens (id, off, vt), maps and itineraries (map, way).

Alice wants a way to Bob (way_{ab}) and agrees to give her name (n_a), her social network information (fid_a, fp_a) and Bob's name (n_b). Using a service composition algorithm, *e.g.,* based on chaining between service inputs/output one would obtain that one must first call _:gps to get Alice's position, _:info and then F:gps to get Bob's one, and finally M:way to get an itinerary from Alice to Bob.

Solving out broken compositions. Let us now suppose that Alice moves and looses the GPS signal. The found composition gets broken (no _:gps service).

A first solution is **replacement**, *i.e.,* replacing a broken (disappeared, faulty, or with a bad QoS) service by another one. There are different solutions for this, a typical one is that service s' can replace service s if it produces more outputs using less inputs. Replacement is an efficient technique as far as computation time is concerned (one comparison with each available service). It also yields compositions resembling to the original ones, which is desirable both for technical (commitment to used services) and social (transparency of the process) reasons. A limit of replacement is that a broken service often cannot be replaced by a unique one. It is the case in our example. The only other way to get a gps position for Alice is by calling three services, namely U:reg, U:office, and U:ogps. Further, while replacement can deal with broken services, it cannot deal with added composition needs since in such a case no service is there to be replaced.

A second solution, that deals both with broken services and new needs is **recomposition**, *i.e.,* computing a new composition. However, this means rebuilding models used in the underlying composition technique, while parts of them may still be valid. *Substitution* is often used as a generic term for the way to react to some service that has to be replaced. Technically, in case of 1-1 substitution we will refer to it as replacement, while in case of 1-n substitution

we will refer to it as recomposition (causal input/output relation between the n services have to be computed). It should be noted that recomposition goes further, since computing a brand new composition may result in using a completely different set of services and hence would correspond to a kind of n-m substitution.

Contributions. To correct broken service compositions, we propose **repair** as an alternative solution that goes beyond the limits of service replacement while avoiding recomposition. This technique aims not only at keeping most of the above mentioned models as-is (*i.e.,* not recompute them), but also take benefit from them while computing a corrected composition. As such, repair is a form of *heuristic and guided partial recomposition.* 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 recomposition while retrieving solutions of the same quality. Setting up composition in the AI planning domain, we propose to apply plan repair principles [22] to ASC. We evaluate and compare our repair algorithm with reference to planning-based recomposition, *i.e.,* replanning, using both a case study and a data set generated with a standard ASC benchmark framework. Our approach is completely tool-equipped, including going beyond models, *i.e.,* reading ontologies and service descriptions files and generating WS-BPEL orchestrations.

Organization. Preliminaries on AI planning are given in Sect. 2. Thereafter, our formal models and the application of AI planning to ASC are given in Sect. 2. Our repair algorithm and its principles are presented in Sect. 4. Details on our prototype implementation and experimental comparative evaluation of repair *vs.* recomposition are given in Sect. 5. Related work is presented in Sect. 6 and we end up with conclusions and our perspectives in Sect. 7.

2 Preliminaries

In this section we introduce AI planning [12]. It been applied with success to service composition [25,7], among others due to its support for under-specified composition requirements which are well-suited to end-user composition.

Definition 1. *Given a finite set* $L = \{p_1, \ldots, p_n\}$ *of proposition symbols, a* planning problem *[12] is a triple* $P = ((S, A, \gamma), s_0, g)$*, where:*

- $S \subseteq 2^L$ *is a set of states.*
- A *is a set of actions, an* action *a being a couple* $(pre, effects^+)$ *where* $pre(a) \subseteq L$ *and* $effects^+(a) \subseteq L$ *denote respectively the preconditions and the (positive) effects of* a*.*
- γ *is a state transition function such that, for any state* s *where* $pre(a) \subseteq s$*,* $\gamma(s, a) = s \cup effects^+(a)$*.*
- $s_0 \in S$ *and* $g \subseteq L$ *are respectively the initial state and the* goal*.*

The definition in [12] takes into account predicates and constant symbols which are then used to define states (ground atoms made with predicates and constants). We directly use propositions here. It also includes negative effects of actions. Since we do not use them in our approach we remove them for clarity.

A *plan* is a sequence of actions $\pi = a_1; \ldots; a_k$ such that $\exists s_1 \in S, \ldots, s_k \in S$, $s_1 = s_0$, $\forall i \in [1, k]$, $precond(a_i) \in s_{i-1} \wedge \gamma(s_{i-1}, a_i) = s_i$. Different algorithms have been proposed to solve planning problems and get plans from them, *e.g.,* depending on whether they are building the underlying graph structure in a forward (from initial state) or backward (from goal) way. We propose to use an algorithm based on planning graphs [4] since they yield a compact representation of relations between actions and represent the whole problem world. Even after some changes, whole parts of planning graphs would then still be valid. Moreover, recent works have demonstrated the suitability of this model for ASC [3,32].

A planning graph G is a directed acyclic leveled graph (see, Fig. 2). The levels alternate proposition levels P_i and action levels A_i. The initial proposition level P_0 contains the initial propositions (s_0). The planning graph is constructed from P_0 using a polynomial algorithm. An action a is put in layer A_i iff $pre(a) \subseteq P_{i-1}$ and then $effects^+(a) \subseteq P_i$. The planning graph actually explores multiple search paths at the same time when expanding the graph, which stops at a layer A_k iff the goal is reached ($g \subseteq A_k$) or in case of a fixpoint ($A_k = A_{k-1}$). In the former case there exists at least a solution, while in the later there is not. Solution(s) can be obtained using backward search from the goal. Planning graphs whose computation has stopped at level k enable to retrieve all solutions up to this level (while other planning techniques are only able to retrieve a single one). Additionally, planning graphs enable to retrieve solutions in a concise form, taking benefit of actions that can be done in parallel (denoted with ||).

3 Models

A service signature is made up of a set of operations which can be described in terms of their inputs and outputs. Still, service being developed by different third-parties, one can hardly imagine, and further, achieve service interoperability at the service signature level. Semantic annotations help in solving this issue [27]. Therefore, we associate semantic information to inputs and outputs.

Definition 2. *Given a set D of concepts, a service w is a set Op of operations where for each o in Op, $in(o) \subseteq D$ (resp. $out(o) \subseteq D$) denote the inputs (resp. the outputs) of o.*

Our service model can be related to WSDL, with semantic annotations for inputs and outputs described using SAWSDL and ontologies supporting annotations described using OWL. Most Web services currently posted online are stateless black boxes (no conversations). For example, numerous services listed by `webservicelist.com` and `xmethods.net` are this kind of Web services, with capabilities that range from checking stock prices, weather, or driving directions to calculating currency exchanges, mortgages, etc. Therefore, for each service w with n operations $o_1, \ldots, o_n$ we may do as if we had n services $w : o_1, \ldots, w : o_n$.

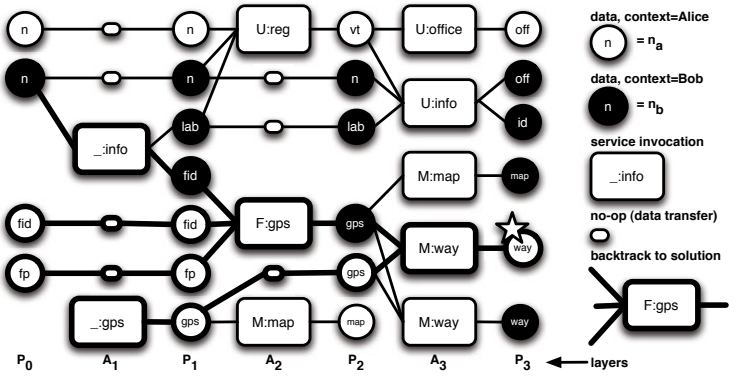


Fig. 2. Planing graph for WSC_1=({services in Fig. 1, right}, {n_a,n_b,fid_a,fp_a}, {wayd_{ab}})

Definition 3. *A service composition problem is a triple* $(W, D_{\text{in}}, D_{\text{out}})$*, where* W *is a set of services,* D_{in} *are provided inputs, and* D_{out} *are expected outputs.*

Following [32], it is possible to map a service composition problem (W, D_{in}, D_{out}) to a planning problem $P = ((S, W, \gamma), D_{\text{in}}, D_{\text{out}})$ with service inputs being mapped to action preconditions ($in(w) \mapsto pre(w)$) and outputs to positive effects ($out(w) \mapsto effects^{+}(w)$). Plans can be encoded in any orchestration language with assignment and sequence operators, *e.g.,* WS-BPEL. Additionally, planning graphs enable to retrieve plans with parallel invocations. These can be encoded using parallel operations (WS-BPEL flow).

Application. The planning graph for our example is given in Fig. 2. To keep figures legible, in a layer P_i we draw only data that is used in some layer $A_{j,j>i}$. We do not draw a service in a layer A_i if it is already in some layer $A_{j,j<i}$, unless if its input data can be re-generated meanwhile. Finally, in the sequel we will not take into consideration the fact that M:way can be used in layer A_3 to produce way_{ba} (it is dual to it producing way_{ab}). Backtracking from the goal, we get a plan, (_:info || _:gps) ; F:gps ; M:way, which can flattened into two sequential plans: _:info ; _:gps ; F:gps ; M:way and _:gps ; _:info ; F:gps ; M:way.

4 Repairing Service Compositions

4.1 Change Modelling

Service composition should be considered in a world that is subject to change. Accordingly, a service composition problem model, $(W, D_{\text{in}}, D_{\text{out}})$, should be updated to accommodate inputs (D_{in}), goals (D_{out}) and service (W) changes.

Goals may change, some getting out of interest while new ones may appear. We denote them respectively with D^{-}_{out} and D^{+}_{out}. Services become unavailable due to many reasons, *e.g.,* network failure or user mobility, and accordingly new

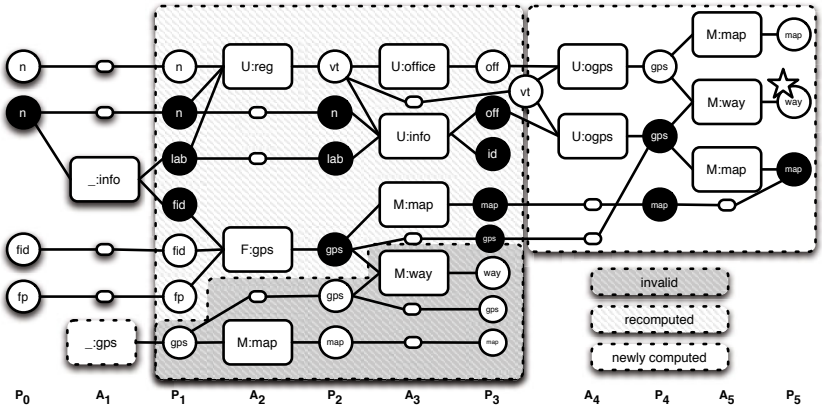


Fig. 3. Planing graph for WSC_1 after removal of _:gps (recomposition)

services may appear. We denote them respectively with W^- and W^+. We may then set $D'_{\text{out}} = D_{\text{out}} \backslash D^-_{\text{out}} \cup D^+_{\text{out}}$ and $W' = W \backslash W^- \cup W^+$.

When change has to be considered during execution, we update D_{in} to the state of the partially executed composition. We also update D_{in} if the inputs of the service composition problem change. In other cases, we have $D'_{\text{in}} = D_{\text{in}}$. Finding a solution for an updated problem may require to rollback an executed service, w, and accordingly its effects. We do not constraint how the effects are canceled (operation in the same service or not), but we set $D'_{\text{in}} = D_{\text{in}} \backslash effects^+(w)$.

With this modelling, change yields a new service composition problem, $(W', D'_{\text{in}}, D'_{\text{out}})$, and, in turn, a new planning problem, $P' = ((S', W', \gamma'), D'_{\text{in}}, D'_{\text{out}})$.

Application. In our example we have $D'_{\text{in}} = D_{\text{in}}$, $D^-_{\text{out}} = D^+_{\text{out}} = W^+ = \emptyset$, and $W^- = \{\text{\_:gps}\}$. Using recomposition on the planning graph we get Fig. 3. One can see that an important part of the graph is rebuilt, before a new part is grown and yields a composition solution, either _:info ; (U:reg || F:gps) ; U:office ; U:ogps ; M:way (way between the building next to Alice and Bob's position), or _info ; U:reg ; (U:office || U:info) ; (U:ogps || U:ogps) ; M:way (itinerary between the building next to Alice and Bob's office).

4.2 Repair Algorithm

Broken preconditions and broken plans. After impacting change, we get a partial planning graph G with a set of *Broken Preconditions* $BP^G_{m\in[1,n]}, BP^G_m \subseteq P_i$. BP^G_n are unsatisfied goals, while $BP^G_{m\in[1,n-1]}$ are inputs of actions in A_{m+1} that are no longer available. Let A be a set of actions, we denote $out(A)$ (resp. $in(A)$) the set $\bigcup_{a\in A} out(a)$ (resp. the set $\bigcup_{a\in A} in(a)$). We have $BP^G_{m\in[1,n-1]} = \{p \in in(A_{m+1}) | p \notin D_{\text{in}} \bigcup_{k\in[1,m]} out(A_k)\}$ and $BP^G_n = \{p \in D_{\text{out}} | p \notin D_{\text{in}} \bigcup_{k\in[1,n]} out(A_k)\}$. We may also focus on a given plan, say π. Let π_i be the set of actions in π at step m. Due to π computation with the planning graph we have

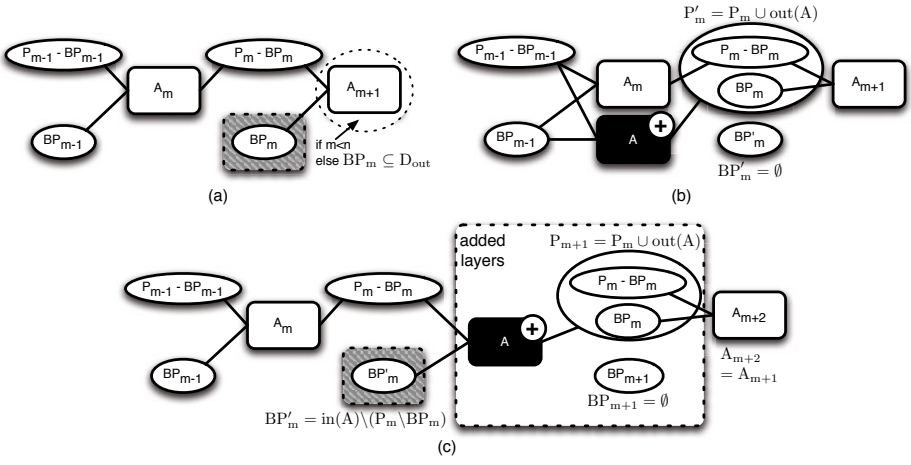


Fig. 4. Broken preconditions at level m, $m \leq n$ (basic repair)

$\pi_m \subseteq A_m$. We have then broken preconditions related to π, $BP^{\pi}_{m\in[1,n-1]} = \{p \in in(\pi_{m+1}) | p \not\subseteq D_{\text{in}} \bigcup_{k\in[1,m]} out(\pi_k)\}$ and $BP^{\pi}_n = \{p \in D_{\text{out}} | p \notin D_{\text{in}} \bigcup_{k\in[1,n]} out(\pi_k)\}$. Note that BP^G_m and BP^{π}_m are incomparable.

We then proceed as follows. If π is not broken ($\forall m \in [1,n]$, $BP^{\pi}_m = \emptyset$) we have nothing to do. π is still a valid solution. Else, we run our algorithm for BP^{π} (hence, in the sequel, we will use BP_m for BP^{π}_m for simplicity). Note that if π is broken but G is not ($\forall m \in [1,n]$, $BP^G_m = \emptyset$) then there is still at least a solution in G, which we may retrieve using backtracking. However, this may yield a solution which is very different from π. Running our algorithm we try first to get a resembling solution. If this fails, any other solution will be found too by the algorithm.

Repair principles. For a proposition level where BP_m is not empty (Fig. 4 (a)), we search for candidate services A which can produce BP_m and insert them into action level m (Fig. 4 (b)). This promotes shorter repair solutions. Sometimes, the lower proposition level P_{m-1} does not contain all the inputs needed by A. We then insert A into a new action level $m+1$ (Fig. 4 (c)). By doing this, we can use more propositions since $P_{m-1} \subseteq P_m$ but increase the plan length by one.

This technique can fail if we cannot find a set of services A that produce all the broken preconditions in BP_m. As a solution we may degrade our basic repair principle. If it is not the goal layer ($m < n$), we remove the unrepairable part (Fig. 5). New broken preconditions may appear at level $m + 1$ and will be treated in a next iteration. If it is the goal layer ($m = n$), then there is no solution, neither with repair nor with recomposition. Degraded repair increases computation time wrt. basic repair since it gets closer to recomposition. Still, it enables to find a solution whenever it exists.

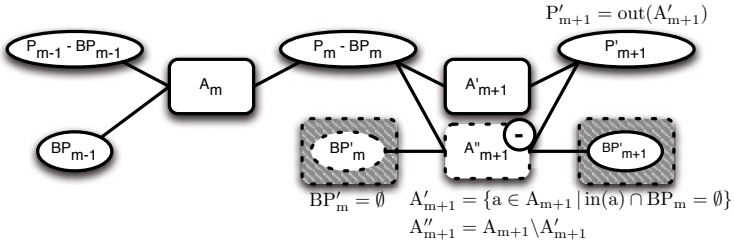


Fig. 5. Broken preconditions at level m, $m < n$ (degraded repair)

Repair algorithms. The main algorithm is the graph repair algorithm, Algorithm 1. It corresponds to the repair principles that are applied for each layer with broken preconditions, starting with the upper layer first. Once we get a repaired graph, a solution is computed with backward search, as with replanning.

Algorithm 1. G'= Repair(G,W)

inputs: partial planning graph G with size n, available services W
outputs: repaired planning graph or `fail`

while $\exists BP_i, i \in [1,n]$, $BP_i \neq \emptyset$ **do**
  $m = \max\{i \in [1,n] | BP_i \neq \emptyset\}$;
  $candidates = \{w \in W | out(w) \cap BP_m \neq \emptyset\}$;
  if $BP_m \subseteq out(candidates)$ **then** {//fixable}
    $\{A, C\}$ = SelectServices($candidates$, G, BP_m, m);
    if $C = \emptyset$ **then** {//insert A into action level m}
      $A_m = A_m \cup A$; $P_m = P_m \cup out(A)$; $BP_m = \emptyset$;
    else {//add A into action level $m+1$}
      $\{A, C\}$ = SelectServices($candidates$, G,BP_m, $m+1$);
      insert a new proposition level P_{m+1} and a new action level A_{m+1};
      $A_{m+1} = A$; $P_{m+1} = P_m \cup out(A)$; $BP_{m+1} = \emptyset$;
      $BP_m = in(A) \backslash (P_m \backslash BP_m)$;
  else {// not directly fixable}
    if $degradeOption = no \vee m = n$ **then** {// no degradation or last layer}
      return `fail`
    else {//degradation is possible}
      $A_{m+1} = \{a \in A_{m+1} \,|\, in(a) \cap BP_m = \emptyset\}$; $P_{m+1} = out(A_{m+1})$
      $BP_m = \emptyset$; recompute BP_{m+1}

In this algorithm, SelectServices (Alg. 2) is in charge of selecting the best services from candidate services to obtain some broken preconditions.

It is a greedy search algorithm. We use (1) and (2) to select the best services. The evaluation depends on the layer where w is to be added. At the highest level n, we define:

$$f1(w,G,m) = \frac{|BP_n \cap out(w)|}{\max_{\Omega}} \times 2 + \frac{|in(w) \cap P_{m-1}|}{\max_{Pi}} - \frac{|in(w) - P_{m-1}|}{\max_{Pe}}, \text{if } m = n; \quad (1)$$

Algorithm 2. $\{W_{\text{selected}}, BP_{\text{new}}\}$ = SelectServices($candidates$, G, BP_{old}, m)

inputs: candidate services $candidates$, planning graph G whose highest level is n, broken preconditions BP_{old}, level m where services to be added
outputs: selected services W_{selected}, new broken precondition BP_{new}

1: $W_{\text{selected}} = \emptyset$; $BP_{\text{new}} = \emptyset$;
2: **while** $BP_{\text{old}} \neq \emptyset$ **do**
3: $w \in candidates$ is the best services by (1) or (2), depending on m;
4: $candidates = candidates \backslash w$; $W_{\text{selected}} = W_{\text{selected}} \cup \{w\}$;
5: $BP_{\text{old}} = BP_{\text{old}} \backslash out(w)$; $BP_{\text{new}} = BP_{\text{new}} \cup in(w) \backslash P_{m-1}$;
6: **return** $\{W_{\text{selected}}, BP_{\text{new}}\}$;

where $|BP_n \cap out(w)|$ is the number of unimplemented goals that can be implemented by w; $|in(w) \cap P_{m-1}|$ is the number of w inputs that can be provided by propositions in P_{m-1}; $|in(w) - P_{m-1}|$ is the number of w inputs that cannot be provided by propositions in P_{m-1}. This set needs to be added into BP if w is added. $\max_\Omega$, $\max_{Pi}$, and $\max_{Pe}$ are the maximum value of the nominators in the neighborhood respectively to normalize each term. The first term is given a weight of 2 to increase its significance.

If w is to be added in another layer ($m < n$) then the evaluation function is:

$$f2(w, G, m) = f1(w, G, m) + \frac{\sum_{i \in [m,n]} |BP_i \cap out(w)|}{\max_{BP}}, \text{if } m < n; \qquad (2)$$

Compared to (1), the added term ($\sum_{i \in [m,n]} |BP_i \cap out(w)|$) is the number of the broken propositions in the level P_m and above that can be satisfied by w outputs. $\max_{BP}$ is the maximum value in the neighborhood to normalize each term. For clarity, we have not put selection exception cases. To ensure repair termination, we do not select services that would reproduce the same set of broken preconditions ($B = C$), nor do we select a service that has been removed in degraded mode.

According to [4], the size of the full planning graph is polynomial to the size of services and the size of propositions. In the worst case, Algorithm 1 explores all the possible paths in the graph, therefore it terminates in polynomial time.

Application. The application of repair to our example is given in Fig. 6. We begin (a) with $BP_2 = \{\mathsf{gps}_a\}$. We select $A = \{\mathsf{U:ogps}\}$ but since $in(A) = \{\mathsf{off}_a\} \not\subseteq P_1$ we add a new level and add A in the new action layer (b). Still, BP_2 (now $\{\mathsf{off}_a\}$) is not empty. We then select $A = \{\mathsf{U:office}\}$ and since $in(A) = \{\mathsf{vt}_a\} \not\subseteq P_1$ we add again a new level (c). We then observe that $BP_2 = \emptyset$ since $in(A) \subseteq P_2$. There is no more any $BP_i \neq \emptyset$ so repair has succeeded. We find the solution which is the closest to the original one, namely _:info ; (U:reg || F:gps) ; U:office ; U:ogps ; M:way. As one can see, the new graph is grown *on top* of the existing one and computed parts are smaller than with recomposition (Fig. 3).

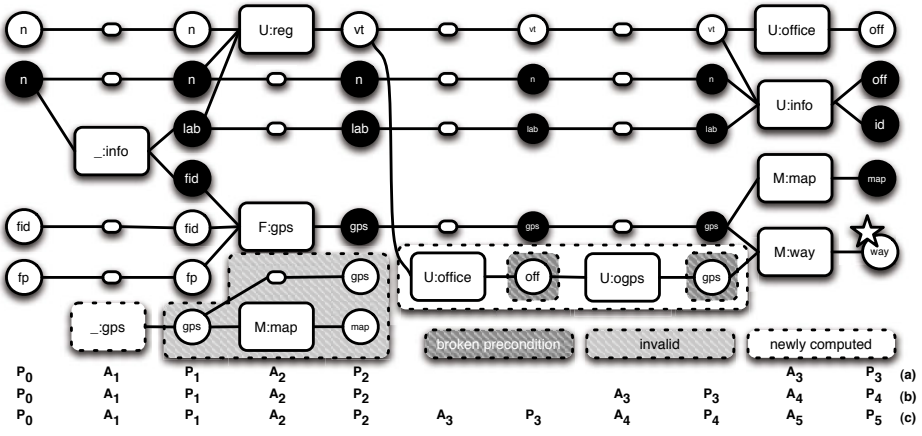


Fig. 6. Planing graph for WSC_1 after removal of _:gps (repair)

5 Implementation and Experimental Evaluation

5.1 Evaluation Criteria

In order to evaluate efficiency and quality of repair *vs.* recomposition (replanning), we use four criteria. *Composition Time* is the time to get the first feasible solution or report nonexistence of it. *Time Steps* and *Number of Services* are respectively the number of layers and the number of services used in the solution. Smaller values lead to more parallelism and less invocations, hence more efficiency at execution time. We assume here that all services have the same execution cost. A perspective is to take into account QoS both in the service model and our heuristic service selection. Finally, *Plan Stability* [9] given two plans π and π' is defined as the number of actions in π not in π' plus the number of actions in π' not in π. Stability is beneficial for transparency and for keeping commitment to services.

5.2 Implementation and Benchmark

The PGA tool. Our approach is fully automated thanks to a tool we have implemented in Java, PGA (Planning Graph with Adaptation). PGA includes three main algorithms. The first one is a Java implementation of the standard graph planning algorithm [4]. It is used to build the original planning graph for a service composition problem and to find a solution for it (or detect there is none). The second algorithm is used to apply change on a planning graph. One may then either call the first algorithm to perform replanning, or use the third one, which implements our repair technique. As far as input files are concerned, PGA can read an OWL file describing ontology concepts and a set of Web service interfaces described in WSDL. Note that the later are annotated with a simple extension mechanism instead of using full-fledged SAWSDL. PGA is also connected to the

WSC platform[1] which is used to generate WS-BPEL orchestrations from XML descriptions of compositions.

Experimental benchmark. PGA has been used to generate solutions for our example (represented in Fig. 3 and Fig. 6). However, this example contains only 20 ground concepts (gps_a, gps_b, n_a, n_b, etc.) and 18 operations (corresponding to the operations in Fig. 1, with duplicates due to genericity, *e.g.*, we have two M:map operations, one for gps_a and one for gps_b). To perform comparison of recomposition and repair, we needed a larger benchmark.

The WSC testset generator is a tool that enables to compare service composition algorithms by generating sets of semantically annotated service interfaces. We use a data set with 351 services. The services can use in their input and output messages parameters in a list of 2891 parameters which are from 6209 instances of 3081 semantic concepts. Given a solution depth, the data generator generates several groups of solutions, each of which has the given solution depth. The solution groups do not share services. Within a group, some services can directly substitute others as they use the same input set and produce the same outputs. The generator randomly generates a lot of "padding" Web services around the services used in solutions. These "padding" services do not have the outputs that can be used by the services within a solution. The data set has four solution groups (1–4) with respectively 9, 18, 19, and 27 levels.

5.3 Experiment Results

We present here the comparative evaluation made for repair and replanning in case of service removal and with group 1 (initial solution: 9 layers and 10 services) Each point is obtained from the average of 100 independent runs. We remove different percentage of services from the service set. We stop at 30% since after it the success rate both for repair and replanning falls down below 50%.

Figure 7(left) shows that our repair algorithm is faster than replanning. Figure 7(right) compares the quality of the solutions. We can see that repair retrieve solutions with the same quality than replanning (plots at superposed).

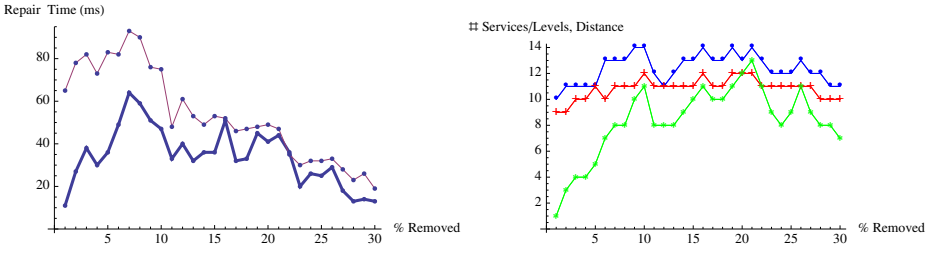


Fig. 7. Repair *vs.* replanning – composition time (thick: repair, thin:replanning) and quality (same for repair/replanning, •/blue: services, +/red: levels, ∗/green: distance)

[1] `http://ws-challenge.georgetown.edu/wsc09/software.html`

6 Related Work

Service composition has been studied under various assumptions and models [27,8,16]. Among these, planning has successfully been used to support underspecified composition requirements [25,7]. Still, to quote [24], service composition is today largely a static affair. In this paper we address the evolution and adaptation of composition under changes.

When the issue is to react to a disappeared or faulty service, or to a service with a bad QoS, strict *replacement* can be used, *e.g.,* [10,13,6] for some recent works. Some works support a less strict notion of replacement, *e.g.,* with transformers to solve out message mismatching [20]. Replacement is limited to 1-1 substitution. Further, it focuses on finding replacement for broken services, while solving a broken composition may require removing other ones. Supporting the substitution of one service by several ones, and using repair degradation, our algorithm has therefore higher success rate than replacement.

Going further than 1-1 substitution, supporting both 1-n substitution and added needs, can be done with *recomposition*, *i.e.,* running a composition algorithm on an updated composition problem. Any algorithm defined for service composition [27,8,16] would apply there, including many ones based on some form of planning, *e.g.,* using heuristic search [19,23], linear programming [31], automated reasoning [14,29]. In our previous works we have proposed to build on planning graphs [32,3]. In this paper we have thoroughly compared our repair algorithm wrt. graph planning recomposition. We get solutions of the same quality in less computation time.

A complementary domain of research is *software adaptation*, devoted to the generation of *mediators* (also called *software adaptors*, or simply adaptors) to solve mismatch between components or services [28]. Software adaptation can be used as a composition technique (when adapting between the composition need and available services). It can also be used as a repair technique that keeps the original (broken) composition unmodified and builds a mediator in between this and available services. This technique has been employed for 1-1 replacement (adaptation between one client and one server), *e.g.,* in [5,21]. Some other works have applied software adaptation in the large between n services, *e.g.,* [17,1]. Computing a mediator is as expensive as computing a composition. Our repair technique then share the same benefits wrt. software adaptation than wrt. recomposition. Moreover, n-ary adaptation applies to services with conversations, hence it requires that the services to adapt are previously discovered (which is complex in case of conversations). Our repair technique does not require this since it takes the assumption that services have no conversation.

Composition repair and reuse has been proposed for fault recovery. Substitution and compensation actions are predefined in [10]. Whenever an erroneous condition is encountered, the execution of the process is stopped and a repair plan generated by automated reasoning is inserted and executed. [18] proposes to substitute the faulty services and rebind to new services, or completely recompose a solution. Still, [18] does not present specific solutions to these strategies. [2] proposes a language to define the substitution actions and uses them

in a self-healing mechanism implemented in the JBOSS rule engine. In workflow management, people also study how to dynamically handle failures. For example, [11] proposes mechanisms like termination of failed activity instances and replanning. Considering possible new opportunities is another motivation for composition evolution. [15] alternates planning and execution of the composition in order to adapt to new opportunities. Compared the above work, our method can be used for both fault recovery and plan evolution.

Plan Repair has been introduced in AI planning. [22] claims that repair is not better than planning in the worst case, still in many cases the new planning problem resembles the original one, hence repair is a technique that works in practice. We have proposed a preliminary service composition repair technique in [30]. But for its objective (repair *vs.* recompose), it is not a direct ancestor of the approach we have presented here. The [30] algorithm degraded into backward search whenever a service without a possible (1-1 or 1-n) substitution at the same layer was removed. Also, the service selection process could lead to dead-ends, *i.e.,* repair failure while there was indeed a solution to the broken composition. Using layer insertion and repair degradation we are able to avoid these drawbacks. Performing comparison, using WSC generated testsets, between the [30] algorithm and the one presented here, we achieve a stable quality of the solutions but a better success rate and shorter composition time.

7 Conclusions and Future Work

Automatically generated service compositions may get broken upon change in their envisioned environment, *e.g.,* upon service disappearance, service failure, or added needs. We have addressed this issue with a repair approach based on planning graphs, an alternative to replacement and to recomposition. Our approach is completely automatic and tool-equipped. Repair does as good as replacement when 1-1 substitution is possible, but goes beyond this limit, supporting 1-n/n-m substitution and added needs. Empirical evaluation has shown that repair gets solutions of the same quality than recomposition in better computation time. Repair may sometimes fail to find a solution while one is found with recomposition. To overcome this issue, we have proposed degradation techniques in our repair algorithm. Experiments have shown that in practice we still get better computation time with repair than with recomposition.

So far, we have compared our repair algorithm with the standard planning graph algorithm. It would be interesting to further compare our repair technique with more recomposition algorithms, especially those in AI replanning studies. In this paper we took into account that many services do not expose a conversation. A promising direction is to support more expressive models [16] with both service conversations and a rich conversation-based requirement language. We made a first step in this direction with a technique based on planning graphs for the composition of services with conversations in [26]. QoS models could also enrich our heuristic service selection process. By now, our PGA tool is only used to repair broken compositions at the model and process level, not for running

instances. The next step is then to study its integration in an existing runtime monitoring and adaptation framework for services composition such as [20].

Acknowledgement. This work is supported by project "Building Self-Manageable Web Service Process" (RGPIN/298362-2007) of Canada NSERC Discovery Grant, and by project "PERvasive Service cOmposition" (ANR-07-JCJC-0155-01, PERSO) of the French National Agency for Research.

References

1. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service Interaction: Patterns, Formalization, and Analysis. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 42–88. Springer, Heidelberg (2009)
2. Baresi, L., Guinea, S., Pasquale, L.: Self-healing bpel processes with dynamo and the jboss rule engine. In: Proc. of ESSPE, pp. 11–20 (2007)
3. Beauche, S., Poizat, P.: Automated Service Composition with Adaptive Planning. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 530–537. Springer, Heidelberg (2008)
4. Blum, A.L., Furst, M.L.: Fast Planning through Planning Graph Analysis. Artificial Intelligence Journal 90(1–2), 281–300 (1997)
5. Brogi, A., Popescu, R.: Automated generation of bpel adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
6. Cavallaro, L., Nitto, E.D., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009)
7. Chan, K.S.M., Bishop, J., Baresi, L.: Survey and comparison of planning techniques for web service composition. Tech. rep, Dept Computer Science, University of Pretoria (2007)
8. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web and Grid Services 1(1), 1–30 (2005)
9. Fox, M., Gerevini, A., Long, D., Serina, I.: Plan Stability: Replanning versus Plan Repair. In: Proc. of ICAPS, pp. 212–221 (2006)
10. Friedrich, G., Ivanchenko, V.: Model-based repair of web service processes. Tech. Rep. 2008/001, ISBI research group, Alpen-Adria-Universität Klagenfurt (2008)
11. Gajewski, M., Momotko, M., Meyer, H., Schuschel, H., Weske, M.: Dynamic failure recovery of generated workflows. In: Proc. of DEXA Workshops, pp. 982–986 (2005)
12. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers, San Francisco (2004)
13. Grigori, D., Corrales, J.C., Bouzeghoub, M.: Behavioral matchmaking for service retrieval: Application to conversation protocols. Inf. Syst. 33(7-8), 681–698 (2008)
14. Hashemian, S.V., Mavaddat, F.: A logical reasoning approach to automatic composition of stateless components. Fundam. Inform. 89(4), 539–577 (2008)
15. Lazovik, A., Aiello, M., Papazoglou, M.P.: Planning and monitoring the execution of web service requests. Int. J. on Digital Libraries 6(3), 235–246 (2006)
16. Marconi, A., Pistore, M.: Synthesis and Composition of Web Services. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 89–157. Springer, Heidelberg (2009)

17. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)
18. Meyer, H., Kuropka, D., Tröger, P.: Asg - techniques of adaptivity. In: Proc. of AAWS (2007)
19. Meyer, H., Weske, M.: Automated service composition using heuristic search. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 81–96. Springer, Heidelberg (2006)
20. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: Proc. of WWW, pp. 815–824 (2008)
21. Motahari Nezhad, H.R., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development. In: Proc. of WWW, pp. 731–740 (2010)
22. Nebal, B., Koehler, J.: Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. Artificial Intelligence Journal 76(1-2), 427–454 (1995)
23. Oh, S.C., Lee, D., Kumara, S.: Web Service Planner (WSPR): An Effective and Scalable Web Service Composition Algorithm. International Journal of Web Service Research 4(1), 1–22 (2007)
24. Papazoglou, M., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing research roadmap (2006), technical report, `http://infolab.uvt.nl/staff/mikep/publications/`
25. Peer, J.: Web Service Composition as AI Planning – a Survey. Tech. rep., University of St.Gallen (2005)
26. Poizat, P., Yan, Y.: Adaptive Composition of Conversational Services through Graph Planning Encoding. In: Proc. of ISoLA (to appear 2010)
27. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
28. Seguel, R., Eshuis, R., Grefen, P.: An overview on protocol adaptors for service component integration (2008), working Paper from, `http://is.tm.tue.nl/staff/heshuis/publications.html`
29. Sohrabi, S., Prokoshyna, N., McIlraith, S.A.: Web service composition via the customization of golog programs with user preferences. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 319–334. Springer, Heidelberg (2009)
30. Yan, Y., Poizat, P., Zhao, L.: Repairing service compositions in a changing world. In: Proc. of SERA (2010)
31. Yoo, J.W., Kumara, S., Lee, D., Oh, S.C.: A Web Service Composition Framework Using Integer Programming with Non-functional Objectives and Constraints. In: Proc. of CEC/EEE. pp. 347–350 (2008)
32. Zheng, X., Yan, Y.: An Efficient Web Service Composition Algorithm Based on Planning Graph. In: Proc. of ICW 2008, pp. 691–699 (2008)