Repairing Service Compositions in a Changing World

Yuhong Yan*, Pascal Poizat†‡ and Ludeng Zhao*
*Concordia University, Montreal, Canada
Email: {yuhong,ludeng.zhao}@encs.concordia.ca
†University of Evry Val d'Essonne, Evry, France
‡LRI UMR 8623 CNRS, Orsay, France
Email: pascal.poizat@lri.fr

***Abstract*—One of the most important benefits of Service-Oriented Computing is to foster the satisfaction of end-user needs through the automatic generation of composite services out of simpler services existing in the user environment. Different approaches have been proposed in the last years to address this issue, *e.g.,* based on model-checking or AI planning. Still, these approaches do not cope with the inherent dynamicity of the service pervasive world, where not only available services, but also user needs, may evolve over time. Setting up service composition in an AI planning framework, we propose in this paper repair techniques enabling service compositions to adapt at run-time, both to service and requirement changes, paving the way for on-demand and sustainable end-user service composition.**

I. Introduction and Motivation

Service-Oriented Computing (SOC) promotes the reuse of loosely coupled and distributed entities, namely services, and their automatic composition into value-added applications. An issue in SOC is to fulfill this promise with the development of models and algorithms supporting composition in an automatic (and automated) way, generating business processes from a set of available services and composition requirements, *e.g.,* descriptions of end-user needs or business goals. Within the context of the widely accepted service-oriented architecture, *i.e.,* Web services, this issue is currently known as the *Web Service Composition (WSC)* problem. WSC has been widely addressed in the past few years [15]. *AI planning* is increasingly applied to WSC due to its support for automatic composition from under-specified requirements [19]. An AI planning based algorithm usually provides a one-time solution, *i.e.,* a *plan*, for a composition request. However, in the real and open world, change occurs frequently. Services may appear and disappear at any time in a unpredictable way, *e.g.,* due to failure or the user mobility when services get out of reach. Service disappearance requires changing the original plan, and may cause some goals to be unreachable. End-user needs or business goals may also change over time. For example, one may want to add sightseeing functionality to a composition when arrived at a trip destination.

The challenge we address in this paper is to define techniques supporting adaptation for a plan as a reaction to changes (available services or composition requirements). Basically, this may be achieved in two distinct ways. The first one is to perform a comprehensive *replanning* from the current execution state of a plan. Another way is trying to *repair* the plan in response to changes, reusing most of the original plan when possible. We believe plan repair is a valuable solution to WSC in a changing world because:

- it makes it possible to retain the effects of a partially executed plan, which is better than throwing it away and having to roll back its effects;
- even if, in theory, modifying an existing plan is no more efficient than a comprehensive replanning in the worst case [23], we expect that plan repair is, in practice, often more efficient than replanning since a large part of a plan is usually still valid after a change has occured;
- commitment to the unexecuted services can be kept as much as possible, which can be mandatory for business/security reasons;
- finally, the end-user may prefer to use a resembling plan resulting from repair than a very different plan resulting from replanning.

Our proposal is based on planning graphs [13]. They enable a compact representation of relations between services and model the whole problem world. Even with some changes, part of a planning graph is still valid. Therefore, we think that the use of planning graphs can be better than other techniques in solving plan adaptation problems. We first identify the new composition problem with the new set of available services and new goals. The disappeared services are removed from the original planning graph and new goals are added to the goal level, yielding a partial planning graph. The repair algorithm "regrows" this partial planning graph wherever the heuristic function tells the unimplemented goals and broken preconditions can be satisfied. Its objective is not to build a full planning graph, but to fast search for a feasible solution, while maximally reuse whatever is in the partial graph. Compared to replanning, the repair algorithm only constructs a part of a full planning graph until a solution is found. It can be faster than replanning, especially when the composition problem does not change too much and a large part of the original planning graph is still valid. In our experiments, we have proved this. Our experiments also show that the solutions from repair have the same quality as those from replanning. However, in some

cases, our repair algorithm may not find existing solutions where replanning would, which is the trade off of its speed.

As far as Web services are concerned, we take into consideration their WSDL interfaces extended with semantic information for inputs and outputs of operations. We suppose services are stateless, therefore we do not consider the internal behavior of Web services. Accordingly, composition requirements are data-oriented and not based on some required conversation. These choices are consistent with many approaches for WSC, *e.g.,* [17] [18] [12]. More importantly, it suits to the Web Service Challenge [4] which enables us to evaluate our proposal on big-size data sets.

The remaining of the paper is as follows. In section II we present our formal models for services (including semantic information), composition requirements, and Web service composition. We also discuss the relations between these models and real service languages. Finally, we introduce graph planning and its application to WSC, and present its use on a simple example. In sections III and IV we respectively address our model for change in a service composition and introduce the evaluation criteria for replanning and repair. Our repair algorithm is presented in section V and is evaluated in section VI. We end up with discussion on related work, conclusions and perspectives.

II. Formal Modelling

A. Semantic Models

In order to enable automatic service discovery and composition from user needs, some form of semantics has to be associated to services. Basically, the names of the services' provided operations could be used, but one can hardly imagine and, further, achieve interoperability at this level in a context where services are designed by different third parties. Adaptation approaches [21] [6] have proposed to rely on so-called adaptation contracts that must be given (totally or partly) manually. To avoid putting this burden on the user, semantics based on shared accepted ontologies may be used instead to provide fully-automatic compositions. Services may indeed convey two kinds of semantic information. The first one is related to data that is transmitted along with messages. Let us suppose a service with an operation providing hotel rooms. It can be semantically described as taking as input semantic information about travelcity, fromdate, todate and username and providing an hotelregistration. The second way to associate semantic information to services is related to functionalities, or capacities, that are fulfilled by services. Provided each service has a single capacity, this can be treated as a specific output in our algorithms, *e.g.*, the abovementioned service could be described with an additional semantic output, hotelbookingcapacity, or we can even suppose its capacity is self-contained in its outputs (here, hotelregistration).

In our approach, semantic information is supported with a structure called *Data Semantic Structure* (Def. 1).

Definition 1: A Data Semantic Structure (DSS) $\mathcal{D}$ is a couple (D, R^D) where D is a set of concepts that represent the semantics of some data, and $R^D = \{R_i^D : 2^D \rightarrow 2^D\}$ is a set of relations between concepts.

The members of R^D define how, given some data, other data can be produced. Given two sets, $D_1, D_2 \subseteq D$, we say that D_2 can be obtained from D_1, and we write $D_1 \rightsquigarrow_{R_i^D} D_2$, or simply $D_1 \rightsquigarrow D_2$, when $\exists R_i^D \in R^D, R_i^D(D_1) = D_2$. Some instances of the R_i^D relations – representing composition, decomposition and casting – together with implementation rules are presented in [3]. Here we propose a generalization of these.

B. Service and Service Composition Models

A Web service can be considered for composition as a function that takes input(s) and returns output(s) (Def. 2). We abstract here from the way how input and output semantic data, called parameters in the sequel, relate to service operations. An example of this where relations are defined between WSDL message types and service model parameters through the use of XPath can be found in [16].

Definition 2: Being given a DSS (D, R^D), a Web service w is a tuple (in, out) where $in \subseteq D$ denote the input parameters of w and $out \subseteq D$ denote the output parameters of w. We denote $in^- \subseteq in$ the set of input parameters that are consumed by the service.

DSS support data transformation using relations in R^D. This enables service composition where mismatch would usually prevent it (see Section II, II-E for examples of these). For the sake of uniformity, being given a DSS (D, R^D), for each R_i^D in R^D, for each $D_1 \rightsquigarrow_{R_i^D} D_2$, we define a data adaptation service $w_{R_i} = (D_1, D_2)$. Such a service can be automatically implemented, either as a service or as a reusable standalone piece of code in any implementation language supporting XPath and assignment such as WS-BPEL.

If the output parameters of a set of services can produce at least one of the input parameters of another service, we say they can be connected (Def. 3).

Definition 3: Assume every parameter in the parameter set $D_W = \{d_1, d_2, \ldots, d_k\}$ is an output parameter of one of the Web services in a set $W = \{w_1, w_2, ..., w_m\}$, *i.e.*, $W = \{w_i \mid \exists d_j \in D_W, d_j \in out(w_i), i = 1, ..., m\}$. If $\{d_1, d_2, \ldots, d_k\} \rightsquigarrow \{d_n\}$, and $d_n \in in(w_n)$, every service in W can be connected to w_n, annotated as $w_i \triangleright w_n$.

Finally, the last input for any WSC algorithm is the description of the composition requirements corresponding to the user needs (Def. 4).

Definition 4: Being given a DSS (D, R^D), a composition requirement is a couple $(D_U^{\text{in}}, D_U^{\text{out}})$ with $D_U^{\text{in}} \subseteq D$ is the set of provided (or input) parameters and $D_U^{\text{out}} \subseteq D$ is the set of required (or output) parameters.

The objective of a WSC algorithm may now be formally described. Given a set of available Web services, a structure (DSS) describing the semantic information that is associated to the services, and a composition requirement, service composition is to generate a connected subset of the services that satisfies the composition requirement (Def. 5).

Definition 5: A composition requirement $(D_U^{\text{in}}, D_U^{\text{out}})$ is satisfied by a set of connected Web services $W = \{w_1, \ldots, w_n\}$ iff, $\forall i \in \{1, \ldots, n\}$:

- $\forall d \in in(w_i), D_U^{\text{in}} \cup out(w_1) \cup \ldots \cup out(w_{i-1}) \rightsquigarrow d$ and
- $\forall d \in D_U^{\text{out}}, D_U^{\text{in}} \cup out(w_1) \cup \ldots \cup out(w_n) \rightsquigarrow d$

A composition requirement is satisfied iff there is at least one set of connected services satisfying it.

C. Models and Languages

The interface (operations and types of exchanged messages) of a Web service is described in a WSDL [26] file. DSSs are a subclass of what can be described using OWL [24], a very expressive language for describing super(sub)-classes, function domains and ranges, equivalence and transition relations, etc. We can extend OWL for other relations as well, *e.g.*, a parameter can correspond to the (de)composition of a set of parameters according to some functions. WSDL can be extended to reference OWL semantic annotations [4]. The SAWSDL standard [25] can also be used for this.

Some service input parameters are *informatic*, and can be therefore used without limitation. However, other service input parameters are *consumable*, meaning that they can be used only once. For example, given an order processing service, orders are consumable: once processed, orders cannot be used by other services. OWL's `rdf:property` can be extended to describe parameters consumability properties.

In this work we suppose services are stateless and do not feature a behavioural description specifying in which ordering operations are to be called. Hence, without loss of generality, we have supposed each service has only one operation. Whenever a service has more than one, we can use indexing, *e.g.*, service w with operations o_1 and o_2 becomes services $w.o_1$ and $w.o_2$.

D. Web Service Composition as Planning

AI planning [10] has been successfully applied to solve the WSC problem through its encoding as a planning problem [19], [15] (Def. 6). The following definitions in this subsection are modified from [10].

Definition 6: A *planning problem* is a triple $P = ((S, A, \gamma), s_0, g)$, where

- S is a set of states, with a *state* s being a subset of a finite set of proposition symbols L, where $L = \{p_1, \ldots, p_n\}$.
- A is a set of actions, with an *action* a being a triple $(precond, effects^-, effects^+)$ where $precond(a)$ denotes the preconditions of a, and $effects^-(a)$ and $effects^+(a)$, with $effects^-(a) \cap effect^+(a) = \emptyset$, denote respectively the negative and the positive effects of a.
- γ is a state transition function such that, for any state s where $precond(a) \subseteq s$, $\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$.
- $s_0 \in S$ is the initial state.
- $g \subseteq L$ is a set of propositions called *goal propositions* (or simply goal).

A *plan* is any sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$, where $k \geq 0$.

The WSC problem can be mapped to a planning problem as follows:

- each service, w, is mapped to an action with the same name, w. The input parameters of the service, $in(w)$, are mapped to the action's preconditions, $precond(w)$, and its output parameters, $out(w)$, are mapped to the action's positive effects, $effects^+(w)$. Consumable parameters, $in^-(w)$ are also mapped to negative effects, $effects^-(w)$.
- the input parameters of the composition requirement, D_U^{in}, are mapped to the initial state, s_0.
- the output parameters of the composition requirement, D_U^{out}, are mapped to the goal, g.

Different planning algorithms have been proposed to solve planning problems, *e.g.*, depending on whether they are building the graph structure underlying a planning problem in a forward (from initial state) or backward (from goal) manner. In our work we propose to use a planning graph [10] (Def.7) based algorithm.

Definition 7: In a planning graph, a *layered plan* is a sequence of sets of actions $\langle \pi_1, \pi_2, \ldots, \pi_n \rangle$, in which each π_i $(i = 1, \ldots, n)$ is independent (see Def. 8). π_1 is applicable to s_0. π_i is applicable to γ (s_{i-2}, π_{i-1}) when $i = 2, \ldots, n$. $g \subseteq \gamma(\ldots(\gamma(\gamma(s_0, \pi_1), \pi_2) \ldots \pi_n)$.

A planning graph iteratively expands itself one level at a time. The process of graph expansion continues until either it reaches a level where the proposition set contains all goal propositions or a fixed point level. The goal cannot be attained if the latter happens first. Definition 8 defines the independence of two actions. Two actions can also exclude each other due to the conflicts of their effects. We can add both independent and dependent actions in one action layer in the planning graph. However, two exclusive actions cannot appear in the same plan. The planning graph searches backward from the last level of the graph for a solution. It is known that a planning graph can be constructed in polynomial time.

Definition 8: In a planning graph, two actions a and b are *independent* iff they satisfy $effects^-(a) \cap [precond(b) \cup effects^+(b)] = \emptyset$, and $effects^-(b) \cap [precond(a) \cup effects^+(a)] = \emptyset$. A set of actions is independent when its actions are pairwise independent.

E. Example

In order to demonstrate our model and our planning algorithm, we give a simple travelling example. Evaluation on benchmark data with a much larger set of services is presented in section VI.

First we introduce the DSS for our example. The concepts we use correspond to:

- users, uinfo (made up of uname and ucity),
- departure and return dates, fromdate and todate,
- departure and return cities (resp. countries), depcity and depcountry (resp. destcity and destcountry),
- travelling alerts, travelalert,

- flight requests, flightreq, and finally,
- registration information for planes and hotels, planereg and hotelreg.

Additionally, different relations exist between these concepts:

- subsumption: from ucity to depcity and from travelcity to destcity,
- decomposition: from uinfo to uname and ucity
- composition: from depcity, destcity, fromdate, and todate, to flightreq.

As explained before, these four relations will be supported by data adaptation services (respectively cast1, cast2, dec1, and comp1) that can automatically be implemented. The repository of available services contains the following services:

- info1({destcity}, {travelalert}),
- info2({destcountry}, {travelalert}),
- c2C({destcity}, {destcountry}),
- plane({flightreq,uname}, {planereg}), and
- hotel({travelcity,fromdate,todate,uname}, {hotelreg}).

Finally, the user request is:
({uinfo,travelcity,fromdate,todate},
{planereg,hotelreg,travelalert}).

Applying our planning algorithm to this example, we get the planning graph in figure 1. Identity links (when some data is kept for the next data layer) are not represented for clarity, but for required parameters (dashed lines).

From this planning graph, backtracking from the required data (in bold), we can compute two solutions (plans), namely (dec1‖cast2);(cast1‖info1‖hotel);com1;plane and (dec1‖cast2);(cast1‖c2C‖hotel);(com1‖info2);plane, where ; represents the sequence of service calls and ‖ service calls in parallel (flow in BPEL). One may note that without the adaptation enabled by DSS relations (cast1, cast2, dec1, and comp1), no composition would be possible at all.

III. Adapt to the Changing World

The WSC problem should be considered in an open world. An open world is always changing. First, Web services appear and disappear all the time (***environment changes***). New services provide new opportunities to achieve some better effects than the original plan. Therefore, we may consider to use new Web services and replace some old ones. Second, during the execution of the process, business goals may change due to shifting business interests (***goal changes***). Hence, new goals may be added and old ones may be replaced. Third, repairing a failed process also generates goal changes, as well as plan changes (***fault caused changes***). We can diagnose the faulty Web services which are responsible for a halting process [27]. We need to eliminate the failure effects, which includes rolling back the executed Web services and find replacement of the faulty Web services.

Environment Changes: Services can be added and removed in the environment. We model it as a simple operation on the service set as in Equation 1.

$$W' = W - W_{disappear} + W_{appear} \tag{1}$$

Goal Changes: We assume the removed goals and the additional goals are given (probably manually generated by user). We just simply update the goals as in Equation 2. We assume that there is no conflict (mutually exclusive propositions) in g. It is possible to automatically generating new business goals. For example, based on user profile, we can suggest sport activities instead of visiting museum for a sport lover to fill a time gap due to meeting cancellation in a travel plan. Generating new goals is not studied in this paper.

$$g' = g - g_{remove} + g_{new\_business} \tag{2}$$

Fault Caused Changes: We can diagnose the faulty Web services as in [27]. The real cause of the exception (the fault) can occur before any symptom (the exception) is observed. When we observe any symptom, the faulty services are already executed. For an identified faulty Web service w, we have some options. Option 1: we can remove the faulty service w, as if the service had disappeared. Option 2: some exceptions are due to format error. This kind of errors can possibly be fixed by finding another service to do format conversion. In this case, we can leave the service unchanged. Option 3: we can identify that the error is only at a certain output parameter p. Then we can project the function of w on the rest of the rest of its outputs, *i.e.*, remove p from $out(w)$, as in Equation 3.

$$out'(w) = out(w) - p \tag{3}$$

When we remove an executed service, *e.g.*, an executed faulty service, it requires to roll back the effects of this service. If a service w needs to roll back, the goals should be updated as in Equation 4. We do not constraint how the effects of a service can be canceled. It can be another service, or a roll back operation of the same service.

$$g = g - effects^{+}(w) + effects^{-}(w) \tag{4}$$

We generally can solve the adaptation problem in two ways **replanning** or **repair**. By **replanning**, we mean that we try to solve a newly constructed planning problem $P' = ((S', A', \gamma'), s_0, g')$, where A' is an updated set of available Web services, g' is a updated set of goals, with S' and γ' changed accordingly, from scratch. By **repair**, we mean that we try to fix the existing, but broken, plans. We consider that planning graph is a good candidate for adaptation, because a planning graph models the whole problem world. If the problem changes, part of the planning graph is still valid. In this paper, we present a repair algorithm that can grow the partial valid planning graph until a solution is found. Below is an intuitive idea to show that repair can be faster. However, the evaluation will be done in a systematic way in the rest of the paper.

Example: The services in the following table are available. a is known, and e is the request.

$A2BC : a \to b, c$	$A2D : a \to d$	$C2E : c \to e$
$D2E : d \to e$	$D2F : d \to f$	$F2G : f \to g$
$F2H : f \to h$	$G2E : g \to e$	$H2I : h \to i$

TABLE I
Available Services

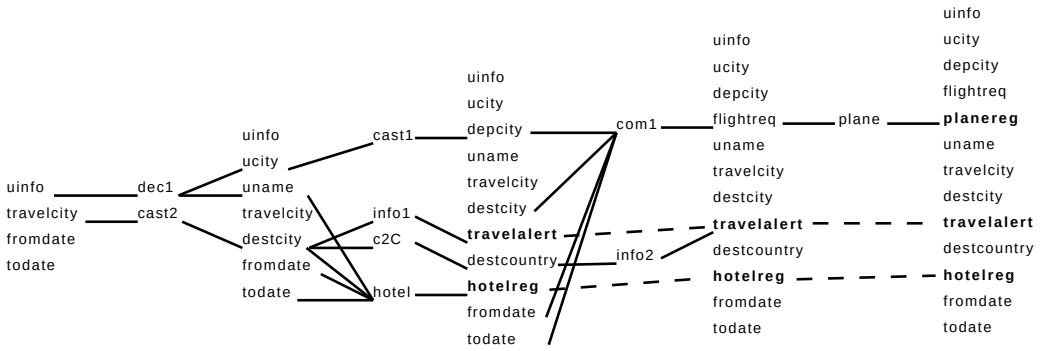

Fig. 1. Travel example - Planning graph

The planning graph is $\{a\} \rightharpoonup \{A2BC, A2D\} \rightharpoonup \{a, b, c, d\} \rightharpoonup \{C2E, D2E, D2F\} \rightharpoonup \{a, b, c, d, e, f\}$.

The solutions are $\{A2BC, C2E\}$ and $\{A2D, D2E\}$. If we remove service $C2E$, we still have another solution. If we remove both $C2E$ and $D2E$, the graph is broken, because e is not produced by any services. We have the following partial planning graph, where e is unsatisfied: $\{a\} \rightharpoonup \{A2BC, A2D\} \rightharpoonup \{a, b, c, d\} \rightharpoonup \{D2F\} \rightharpoonup \{a, b, c, d, \underline{e}, f\}$.

If we do a replanning, a new planning graph is built from scratch: $\{a\} \rightharpoonup \{A2BC, A2D\} \rightharpoonup \{a, b, c, d\} \rightharpoonup \{D2F\} \rightharpoonup \{a, b, c, d, f\} \rightharpoonup \{F2G, F2H\} \rightharpoonup \{a, b, c, d, f, g, h\} \rightharpoonup \{G2E, H2I\} \rightharpoonup \{a, d, c, d, f, g, h, e, i\}$. This graph contains a solution $\{G2E, F2G, D2F, A2D\}$.

The repair algorithm developed in this paper does not build the full planning graph as above, but tries to fix the partial planning graph while searching for a solution. In order to produce the unsatisfied e, we first add a service $G2E$ into the partial graph, which results: $\{a\} \rightharpoonup \{A2BC, A2D\} \rightharpoonup \{a, b, c, d, \underline{g}\} \rightharpoonup \{G2E, D2F\} \rightharpoonup \{a, b, c, d, e, f\}$.

Next, we add a service $F2G$ to satisfy g: $\{a, \underline{f}\} \rightharpoonup \{A2BC, A2D, F2G\} \rightharpoonup \{a, b, c, d, g\} \rightharpoonup \{G2E, D2\underline{F}\} \rightharpoonup \{a, b, c, d, e, f\}$.

We continue this process until the following graph is built: $\{a\} \rightharpoonup \{A2D\} \rightharpoonup \{a, d\} \rightharpoonup \{D2F\}\{a, f\} \rightharpoonup \{A2BC, A2D, F2G\} \rightharpoonup \{a, b, c, d, g\} \rightharpoonup \{G2E, D2F\} \rightharpoonup \{a, b, c, d, e, f\}$.

Now, we have no unsatisfied preconditions and goals. Therefore, we can get a solution $\{A2D, D2F, F2G, G2E\}$. We can see that the repaired graph is simpler than the full planning graph. Our goal is to find at least one solution.

IV. Evaluation Criteria

We want to evaluate the efficiency and quality of replanning and repair in generating new Web service composition solutions. We use three kinds of criteria: execution time, the quality of the new composition, and plan stability.

The composition time: It is the time to get the first feasible solution or report nonexistence of it. It is possible that after removing some services or adding new goals, no solution exists to the new problem. For replanning, since we solve the new problem using Graphplan, we are able to know the existence of solution. For repair, as we solve it using heuristics (see the next section), the algorithm may report no solution in a case where solutions exist.

The quality of the new solution: A solution is better if it invokes less services and if it involves less time steps. For Graphplan, the time steps are the levels of action layers. In one action layer, the services can be executed in parallel. If less services are invoked, the cost of the solution is lower, assuming each service has the same execution cost.

Plan Stability: Plan stability is defined in Defition 9 as in [7]:

Definition 9: Given an original plan, π_0, and a new plan π_1, the difference between π_0 and π_1, $D(\pi_0, \pi_1)$, is the number of actions that appear in π_1 and not in π_0 plus the number of actions that appear in π_0 and not in π_1.

We are interested in achieving plan stability because we prefer that the new plan is similar to the original one. In the Web service composition context, this means that we can keep our promise to the business partners in the original plan.

V. Repair the broken planning graph to generate a solution

Due to the changing world, the available service set and the goal set are updated. In this paper, we are interested in finding a feasible solution fast, rather than an optimal solution. If more services are available or less goals needed to satisfy, the original planning graph can still give a solution. Therefore, we focus on the cases when services are removed or more goals are added, in which cases, the original planning graph becomes only partially valid.

Our idea is to "grow" the partial planning graph again in order to obtain a feasible solution. Our approach is not to build an entire planning graph again, but to fast find a solution. During the "growing" process, we heuristically search into a direction that can satisfy the broken goals and preconditions, as well as making use of the existing partial graph. Our strategy is as follows. Assume

- w: a service whose inputs are $in(w)$ and outputs are $out(w)$;
- G: a partial planning graph;

- $\widetilde{g}$: a set of unimplemented goals;
- BP: a set of unsatisfied preconditions (inputs) of some services in G;
- D_U^{in} is the initial proposal level noted as P_0; the first action level is noted as A_1; the last proposition level P_n which is also the D_U^{out} level.

Assume we want to add an action w to the highest action level n, the evaluation function is:

$$f(G,w) = |\widetilde{g} \cap out(w)| * 10 + |P_{n-1} \cap in(w)| - |in(w) - P_{n-1}| - e(w,G) \quad (5)$$

Where $|\widetilde{g} \cap out(w)|$ is the number of unimplemented goals that can be implemented by w. The coefficient 10 is the weight of this term. It shows that to satisfy the goals is more important than the other needs represented by the following terms;
$|P_{n-1} \cap in(w)|$ is the number of the inputs of w that can be provided by the known parameters at the level P_{n-1};
$|in(w) - P_{n-1}|$ is the number of the inputs of w that **cannot** be provided by the known parameters at the level P_{n-1}. This set needs to be added into BP, if w is added.
$e(w,G)$ is the number of the actions in G that are exclusive with w.

Assume we want to add an action w to action level m, and m is not the goal level, the evaluation function is:

$$f(G,w) = |\widetilde{g} \cap out(w)| * 10 + |P_m \cap BP \cap out(w)| + |P_{m-1} \cap in(w)| - |in(w) - P_{m-1}| - e(w,G) \quad (6)$$

Compared to equation 5, the above equation added term $|P_m \cap BP \cap out(w)|$ which is the number of the broken propositions in level P_m that can be satisfied by the outputs of w.

Algorithm 1 uses heuristics to search for a repaired composition. The algorithm starts from the goal level. It tries to satisfy the unimplemented goals first (Line 2 to 8). When new services are added into the partial planning graph, its precondition may not be satisfied. The unsatisfied preconditions are added to BP to be satisfied at a lower level (Line 8, 17 and 29). This process goes from the goal level toward the initial proposition level (Line 10 to 18). It is possible that after adding actions at A_1, we still have some broken preconditions. In this case, we need to add new levels (Line 19 to 32). We use BPH to record the history of preconditions we have satisfied. If $BPH \cap BP \neq \emptyset$, that means some precondition broken for the second time. It is a deadlock situation. We stop the search.

Algorithm 1 is a best first search algorithm. It does not generate a full planning graph, but rather, to fast fix the broken graph and obtain a feasible solution. It is possible that algorithm 1 does not find a solution to a problem with solutions. However, repair can be faster than replanning, which keeps the quality of solutions (cf. Section VI). It is possible that algorithm 1 generates a graph that contains multiple solutions. Therefore, a `backwardsearch()` function returns the first obtained solution. We know that a solution can be found from the graph made of the originally broken plan and the newly added services. `backwardsearch()` is on this very small graph, thus fast.

Algorithm 1 Search for a repair plan: Repair(W, G, BP, $\widetilde{g}$)

- Input: W, G, BP, $\widetilde{g}$ defined as before
- Output: either a plan or `fail`

1: result = `fail`
2: **while** $\widetilde{g} \neq \emptyset$ **do**
3: select an action w with the best $f(G,w)$ according to equation 5
4: **if** w does not exist **then** break
5: add w to G at action level A_n
6: add $out(w)$ at proposition level P_n
7: remove $\widetilde{g} \cap out(w)$ from $\widetilde{g}$
8: add $in(w) - P_{n-1}$ to BP
9: **if** $\widetilde{g} \neq \emptyset$ **then** return result
10: **for** $m = n-1; m > 0; m--$ **do**
11: **while** $BP \cap P_m \neq \emptyset$ **do**
12: select an action w with the best $f(G,w)$ according to equation 6
13: **if** w does not exist **then** break
14: add w to G at action level A_m
15: add $out(w)$ at proposition level P_m
16: remove $P_m \cap BP \cap out(w)$ from BP
17: add $in(w) - P_{m-1}$ to BP
18: **if** $BP \cap P_m \neq \emptyset$ **then** return result
19: $BPH = BP$
20: **while** $BP \neq \emptyset$, $W \neq \emptyset$ **do**
21: insert an empty proposition level P_1 and empty action level A_1
22: $P_1 = P_0 - BP$
23: **while** $BP \cap P_1 \neq \emptyset$ **do**
24: select an action w with the best $f(G,w)$ according to equation 6
25: **if** w does not exist **then** break
26: add w to G at action level A_1
27: add $out(w)$ at proposition level P_1
28: remove $P_1 \cap BP \cap out(w)$ from BP
29: add $in(w) - P_0$ to BP
30: remove w from W
31: **if** $BP \cap P_1 \neq \emptyset$ **then** break
32: **if** $BP \cap BPH \neq \emptyset$ **then** break **else** add BP to BPH
33: **if** $BP \neq \emptyset$ **then** return result
34: result = backwardsearch(G)
35: return result

VI. Empirical Evaluation

A. Data Set Generation

We use the Web Service Challenge 2009 [2] platform and its tools in our experiments. The platform can invoke the composition algorithm as a Web service and evaluate composition time. The data generator generates ontology concepts in OWL and a set of Web services interfaces in WSDL which use the concepts. Given the number of services and time steps in a solution, the generator algorithm generates a number of solutions around these numbers first, and then randomly generates a set of Web services. A validation tool can verify

the correctness of the result BPEL file by simulating the process to see whether the query parameters are produced. The 2009 challenge supports QoS features which are not used our experiments. We use two data sets (Table II). The first one is relatively small with about 351 available services. The second one is larger with over 4000 available services. In Table II, concepts correspond to the classes defined in OWL. Things are instances of concepts and are used as the parameter names (params) for services. Services are described in a WSDL file. Each service has exactly one port type and each port type has one input message and one output message. Notice that subsumption is modeled in OWL and is used during composition. Negative effects are not modeled in this kind of data sets.

Data Set	1	2
Concepts	3081	3093
Things	6209	6275
Params	2891	45057
Services	351	4131

TABLE II
DATA SETS

B. Implementation and Experiments Set Up

The purpose of this experiment is to compare our repair algorithm with the replanning algorithm which is a standard planning graph algorithm in case of updated problem. The replanning serves as a baseline algorithm. We want to check whether repair algorithm can be better than replanning and, if so, under which conditions. We use four criteria to evaluate the performance of algorithms: composition time, number of services in the plan, levels in the plan, and distance to the original plan. The definition of the criteria can be found in Section IV. We conduct two experiments. In Experiment 1, a certain percentage of available services, from 3% to 24%, is randomly removed. In Experiment 2, a number of services are randomly removed from a solution. The situation of adding new goals is similar to the situation where some goals are not satisfied due to service removal. Thus we only consider service removal in our experiments.

Our implementation includes some technical details that cannot fully be discussed here. First, we apply several indexing techniques for expediting composition process. For each service, we use a hash table to index all the possible concepts (defined in a OWL document) that the service takes as inputs or outputs. The subsumption hierarchy is thus "flattened" in this step so we do not need to consider semantic subsumption during the planning process. Second, we also use a hash table to store the mapping relationships between each semantic concept and all services that can accept that concept as one of their inputs. This is similar to the "reverse indexing approach" introduced in [28]. It allows us to search the invokable Web services from the known concepts very fast - a simple join operation among related rows of the index table instead of checking the precondition of each service in the repository.

C. Results

The following comparison of performance is recorded in the cases that both repair and replanning algorithms can find solutions. Each data point is obtained from the average of five independent runs when both repair and replanning find a solution.

Fig 2 to Fig 5 show the results from Experiment 1. From Fig 2, we can see that the replanning composition time is slightly decreasing when more Web services are removed. It is because the problem is smaller when less Web services are available. However, it is more difficult to repair the plan. Therefore, the repair composition time increases in such a case. However, after a certain percentage (around 20%), the repair composition time descreases. This is because the problem becomes simpler to solve and also because we are less committed to the original composition. Please notice that repair may not find existing solutions. For example, when removing 21% services, we observed 4 failures on 9 runs.

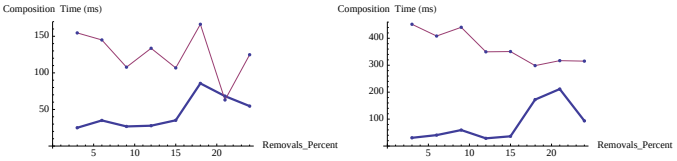

Fig. 2. Composition time with data set 1 (left) and data set 2 (right) (repair - thick line, replanning - thin line)

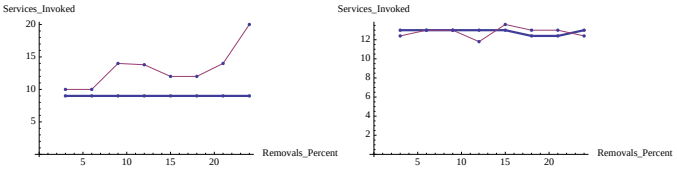

Fig. 3. Number of Services Invoked with data set 1 (left) and data set 2 (right) (repair - thick line, replanning - thin line)

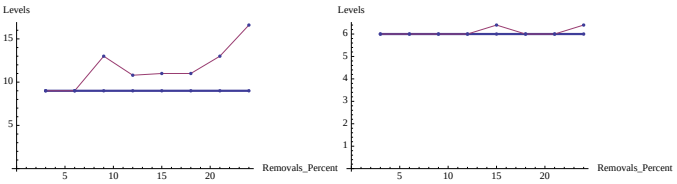

Fig. 4. Number of Levels with data set 1 (left) and data set 2 (right) (repair - thick line, replanning - thin line)

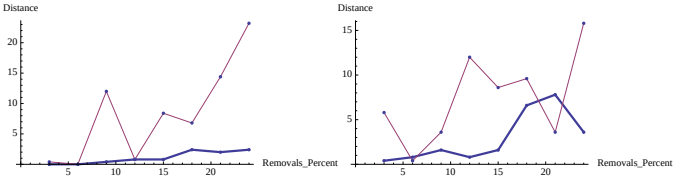

Fig. 5. Distance to the Original Plan with data set 1 (left) and data set 2 (right) (repair - thick line, replanning - thin line)

Fig 3 and Fig 4 show the number of services and the number of levels in a composed plan. The plot for repair is rather flat. Our explanation is that our repair algorithm does not work well when the number of levels are over 10. This is because it is a greedy search algorithm and the successful rate is lower in more level cases. As we do not count unsuccessful cases, we end up showing the cases where the levels are below 10 and very flat. Fig 3 and Fig 4 show that when repair finds a solution, the quality of the solution is pretty good.

Finally, Fig 5 shows that the solution computed with the repair algorithm can be more similar to the original plan than the solution computed with the replanning algorithm.

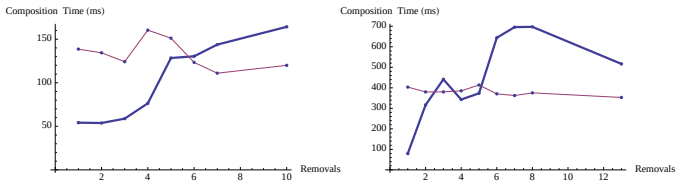

Fig. 6. Composition time with data set 1 (left) and data set 2 (right) (repair - thick line, replanning - thin line)

Fig 6 shows the composition time from Experiment 2. Experiment 2 tries to fix a broken solution (services beings removed from the solution). Since the number of removed services are not big, the composition time for the replanning algorithm does not change a lot for all the runs, because the composition problems on each run are of the same difficulty. When a small number of services are removed, repair is faster than replanning. But this becomes more difficult as more services are removed. Therefore, at one moment, repair takes longer than replanning. However, if services from an original solution are almost completely removed, then repair is like searching for a new plan, which can become faster. This shows in the last data point at the right figure in Fig 6. The comparison of solution quality and distance to original solution is similar as in Experiment 1 and is omitted due to length limitation.

VII. Related Work

Web service composition has been studied under various assumptions and models. If we assume it is not possible to get the internal behavior of a service, we can consider only how to connect the input/output parameters of services. In this case, we can further assume *stateless services* and use *synchronous communication* (request-response style). This is what we call ***data driven problems*** with automatic services.

This WSC problem can be modeled as a planning problem [17] by mapping a service (operation) to an action, the service input parameters as preconditions of the action, and the service output parameters as positive effects of the action. If a parameter can be used only once, it becomes a negative effect. For some composition requirement $(D_U^{\text{in}}, D_U^{\text{out}})$ and a set of actions for services $W = \{w_1, w_2, \ldots, w_n\}$, if one can find a planning structure with D_U^{in} as initial state and D_U^{out} as goal, then the WSC problem can be resolved by the network of W taking D_U^{in} as input and producing D_U^{out} as results. Many planning algorithms can be used to solve the above planning problem, *e.g.,* heuristic search in AI [17] or linear programming [29].

WSC problems also consider semantics defined in languages such as OWL-S and WSMO. A review of some early approaches on planning with description logics can be found in [11]. Due to length limitation, this aspect is not reviewed here.

If we assume services have an internal behavior, *i.e.,* a service is a stateful process, then we need to consider asynchronous communication since we may not get a response immediately after a request message. The response can be produced by some internal nondeterministic process. Usually, the complete internal behavior of services is not exposed. However, behaviors can be partially observed through input and output messages, *e.g.,* using abstract BPEL. We call this ***process driven problems***.

Pistore and his colleagues' work [20] can handle asynchronous communication and the partial observable internal behavior of the candidate services. Their solution is based on belief systems and is close to controller synthesis. The condition to build belief systems is too strong and a large portion of the systems is not controllable (*i.e.,* not deadlock free). Van der Aalst and his colleagues try to solve the same problem [22] using Operation Guide Lines (OGL). While it is often impossible to know the internal behavior of a service, this service may expose its behavior in some way anyway, *e.g.,* in abstract BPEL. An OGL for a service abstracts the service controllable behavior such that, if a partner service can synchronize with the service OGL without deadlocks, then the partner service can also synchronize with the service itself. The difficulty is how to abstract OGL.

The major difference between our work and the above mentioned ones is that we tackle the problem of adapting service compositions to the changing world. This issue is also addressed by replanning in domain independent AI planning. Replanning normally has two phases: a first one determines the parts where the current plan fails and the second one uses slightly modified standard planning techniques to replan these parts [14]. Slightly modified planning techniques normally mean the use of heuristics of recorded decision and paths in searching for actions. In theory, modifying an existing plan is no more efficient than a compete replanning in the worst case [23]. However, we have shown that in some cases, *e.g.,* when the number of services to replace is small wrt. the total number of services used in a composition, repair is a valuable alternative to replanning in terms of computation time.

The DIAMOND project [1] also studies the repair problem. [8] presents a method of composing a repair plan for a process. It predefines substitution and/or compensation actions for an action in a process. [8] is based on a logical formalization which can be processed by modern logical reasoning systems supporting disjunctive logic programming. Similar work can be found in the replanning area as well, for example O-Plan [5]. Whenever an erroneous condition is encountered, the

execution of the plan is stopped and some predefined repair actions are inserted and executed. In contrast, our paper is not about fault diagnosis or exception handling. We change the original plan with available Web services to make it valid again.

VIII. Conclusions and Future Work

Service compositions have to be adapted whenever the composition context, *i.e.*, available services and composition requirements, change. In this paper we have set up service composition in the AI planning domain and we have proposed a planning graph repair algorithm focused at obtaining an updated service composition solution fast rather than obtaining all possible modified service composition solutions. We have proposed composition adaptation comparison criteria, namely composition time, plan quality, and distance between the original and the modified plan. We have then used these criteria to compare our repair algorithm with replanning which is the standard planning technique for plan modification. We observed that when services are removed on a percentage basis of the whole service set, and when available services are many and similar as in Experiment 1, our repair algorithm can be faster than the replanning one. The repaired plan can be as good as the one obtained with replanning but is more similar to the original plan. In the case where one wants to fix an existing plan in which some services are removed, as in Experiment 2, the repair algorithm is faster than replanning only when the number of removed services is small. At a certain point, repair then gets slower than replanning. This suggests there is no silver bullet in adapting plans and that different algorithms should be selected depending on the kind and degree of change.

There are different perspectives to our work. First, we can improve the repair algorithm following existing work in replanning [9] [23] that propose, upon disappearance of some service(s), to remove even more services in order to jump out of local extrema in the search for a new plan. A second perspective is relative to the assumptions we made on services and composition requirements. Taking into account behavioral properties, *e.g.*, stateful services and conversation-based requirements, would increase the expressiveness of our approach as far as Web services are concerned.

Acknowledgment

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

References

[1] Diamond project homepage. http://wsdiamond.di.unito.it/.

[2] Web service challenge 2009 web site. http://ws-challenge.georgetown.edu/wsc09/index.html.

[3] S. Beauche and P. Poizat. Automated Service Composition with Adaptive Planning. In A. Bouguettaya, I. Krüger, and T. Margaria, editors, *Proc. of ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 530–537, 2008.

[4] S. Bleul. Web service challenge rules. http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf, 2009.

[5] B. Drabble, J. Dalton, and A. Tate. Repairing Plans On-the-fly. In *NASA Workshop on Planning and Scheduling for Space*, 1997.

[6] M. Dumas, B. Benatallah, and H. R. Motahari-Nezhad. Web Service Protocols: Compatibility and Adaptation. *Data Engineering Bulletin*, 31(3):40–44, 2008.

[7] M. Fox, A. Gerevini, D. Long, and I. Serina. Plan Stability: Replanning versus Plan Repair. In D. Long, S. F. Smith, D. Borrajo, and L. McCluskey, editors, *Proc. of ICAPS*, pages 212–221. AAAI, 2006.

[8] G. Friedrich and V. Ivanchenko. Model-based repair of web service processes. Technical Report 2008/001, ISBI research group, Alpen-Adria-Universität Klagenfurt, 2008. https://campus.uni-klu.ac.at/fodok/veroeffentlichung.do?pubid=67566.

[9] A. Gerevini and I. Serina. Fast Planning through Greedy Action Graphs. In *Proc. of AAAI/IAAI*, pages 503–510, 1999.

[10] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[11] Y. Gil. Description Logics and Planning. *AI Magazine*, 26(2), 2005.

[12] S. V. Hashemian and F. Mavaddat. Automatic Composition of Stateless Components: A Logical Reasoning Approach. In F. Arbab and M. Sirjani, editors, *Proc. of FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2007.

[13] S. Kambhampati, E. Parker, and E. Lambrecht. Understanding and Extending Graphplan. In S. Steel and R. Alami, editors, *Proc. of ECP*, volume 1348 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[14] S. Koenig, D. Furcy, and C. Bauer. Heuristic search-based replanning. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Proc. of AIPS*, pages 294–301. AAAI, 2002.

[15] A. Marconi and M. Pistore. Synthesis and Composition of Web Services. In *9th International School on Formal Methods: Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 89–157, 2009.

[16] T. Melliti, P. Poizat, and S. Ben Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In *Proc. of FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 146–162, 2008.

[17] S.-C. Oh, D. Lee, and S. Kumara. Web Service Planner (WSPR): An Effective and Scalable Web Service Composition Algorithm. *Int. J. Web Service Res.*, 4(1):1–22, 2007.

[18] S.-C. Oh, D. Lee, and S. Kumara. Flexible Web Services Discovery and Composition using SATPlan and A* Algorithms. In *Proc. of MDAI*, July 2005.

[19] J. Peer. Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen, 2005.

[20] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. of ICAPS*, pages 2–11, 2005.

[21] R. Seguel, R. Eshuis, and P. Grefen. An Overview on Protocol Adaptors for Service Component Integration. Technical report, Eindhoven University of Technology, 2008. BETA Working Paper Series WP 265.

[22] W. M. P. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf. Service Interaction: Patterns, Formalization, and Analysis. In *9th International School on Formal Methods: Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 42–88, 2009.

[23] R. van der Krogt and M. de Weerdt. Plan repair as an extension of planning. In S. Biundo, K. L. Myers, and K. Rajan, editors, *Proc. of ICAPS*, pages 161–170. AAAI, 2005.

[24] W3C. Owl web ontology language overview. http://www.w3.org/TR/owl-features/, 2004.

[25] W3C. Semantic annotations for wsdl and xml schema (sawsdl). http://www.w3.org/TR/sawsdl/, 2007.

[26] W3C. Web services description language (wsdl) version 2.0. http://www.w3.org/TR/wsdl20/, 2007.

[27] Y. Yan, P. Dague, Y. Pencolé, and M.-O. Cordier. A Model-Based Approach for Diagnosing Fault in Web Service Processes. *Int. J. Web Service Res.*, 6(1):87–110, 2009.

[28] Y. Yan, B. Xu, and Z. Gu. Automatic Service Composition Using AND/OR Graph. In *Proc. of CEC/EEE*, pages 335–338. IEEE, 2008.

[29] J.-W. Yoo, S. Kumara, D. Lee, and S.-C. Oh. A Web Service Composition Framework Using Integer Programming with Non-functional Objectives and Constraints. In *Proc. of CEC/EEE*, pages 347–350, 2008.