Forward Expansion and Add Service Implementation Details

Restrictions

A service can be added to a search graph during forward expansion only if it produces at least one parameter as output that does not exist in the prdSet at the time that the service has to be added.

- **Reasons**
    a. **Preventing duplicate services for reducing resource wastage:** Without this restriction, it is possible that the same service, say, w1, gets added first in, say, layer 5, and then in a later iteration in, say, layer 8 in the same search graph. This can happen if some services in layer 7 produce some parameters that can act as inputs to w1. In case this scenario results in a plan that contains both instances of w1, there is a wastage of resources in executing w1 twice for computing the same set of parameters.

    To avoid multiple execution of the same service, an alternative could be to move w1 from layer 5 to 8. However, such a change would affect all the predecessor and successor services of w1 (even requiring the successors to be moved to layers 9 and later). Further, changes to these services would create a cascade effect and ultimately result in a huge and complicated change in the search graph, which makes it a highly non-feasible solution.

    b. **Preventing infinite looping of forward expansion:** In the absence of this restriction, every eligible service in the service repository, irrespective of whether it is already present in the graph, will be validated and added to the search graph in every iteration. Therefore, every iteration will add services to the graph and the termination condition of forward expansion process - when no services are added in an iteration - will never be satisfied, and the process will enter an infinite loop after adding all possible services for a composition request.

    c. **Preventing wastage of effort spent on redundant processing:** Without this restriction, the same services will be validated and added in the same layers (as earlier iterations) of the search graph in every iteration, if not in later layers as described in point (a). This is just redundant processing of the same services which adds a lot of unnecessary effort to the process.

- **Side-effects**
    Some of the possible but longer (containing more atomic services) alternative solutions could get rejected. For example, suppose by the time layer 5 of a search graph is constructed all the requested output parameters have been generated and added to the prdSet. Now, even if a service w1 (with predecessors in layer 5) from the repository is capable of producing one or more of the requested output parameters, it cannot be added to the search graph to later produce an

alternative solution plan unless w1 also produces at least one output parameter that does not exist in prdSet.

Justification: The aim of these algorithms and this thesis is not to obtain all possible solutions or the most optimum solution for composition problem. The aim is to be able to generate a reasonable number of alternative solutions within a reasonable amount of time. The solutions are then to be used for generating a service package, translated into a Lucid program and used to simulate a real-world broker system. Therefore, a tradeoff needs to be achieved between time complexity and solution diversity.

- **Inapplicability**
  This restriction does not apply to a composite service that is constructed using other composite services as atomic units and the repeated service or set of services exists in one of these "atomic" composite services. For example, suppose service w1 exists in composite services cw1 and cw2. Now, cw1 and cw2 both are used as atomic units while constructing another composite service cw3. In this case, w1 gets repeated in cw3, which should violate the restriction above. However, this is not the case because here cw1 and cw2 are used as black-boxes or atomic units and the services inside them are not visible in the search graph of cw3.

  Side-effect and Justification: This repetition of w1 in cw3 causes redundant execution of w1 for performing the same task twice, which results in a wastage of resources. However, in this scenario there could be a possible requirement of multiple execution of the same service within a composite service. For example, micro services, such as an addition service, could be required for adding some numbers in cw1 and some other numbers in cw2. In such a scenario, multiple execution is not a redundant effort but a requirement.

Assumptions

- A composition request is considered invalid and results in failure of the forward expansion process if it can be served by a single service present in the service repository. In this scenario, composition of services is not required and hence there is no need for processing such a request for composition. A simple search of the repository would be sufficient to find the required service.
  At the forward expansion stage, this check is implemented by ensuring that the resulting search graph contains more than one services. However, a search graph is likely to have excess services as well. Therefore, similar checks will be performed during later stages of the service composition process as well to discard requests that can be served by a single service.

- A composition request might include its expected output parameters in its input parameters as well. Such a request could result in an empty search graph or an otherwise unsolvable problem because the prdSet would already include the output parameters. Therefore, the only way a solution could be obtained for such a problem would be through services that produce some additional output parameters along with the requested output parameters. If a valid search graph

can be constructed for such a request, the processing is taken to further stages, otherwise the forward expansion stage fails.
We do not perform a check on the composition request to prevent all output parameters from being present in the input parameters as this is an added early restriction on the service composition model. Rather, we prefer checking the search graph that results from this request for solvability of this problem.

Differences between Algorithm and Implementation

- In the forward expansion algorithm, as soon as a service is added to a search graph, its output parameters are added to the prdSet. However, in the implementation, once an entire layer has been added, the output parameters of all the services in this new layer are added to the prdSet at once (not immediately after adding each individual service).

  Reason: The algorithm approach restricts some of the possible alternative solutions to the problem. For example, consider two services w1 and w2 that have predecessors in layer 2 and produce the same output parameters o1 and o2. By the algorithm approach, once w1 is added to layer 3, o1 and o2 will be added to prdSet. Then, w2 can never be added to layer 3 despite it being a valid candidate service. However, by the implementation approach, o1 and o2 will not be added to prdSet until all possible services have been added to layer 3 thereby allowing w1 and w2 both to be added to layer 3 as alternative solutions.

- In the forward expansion algorithm, the CheckUserConstraints statement is used for verifying the constraints imposed by the requester on the Quality of Service features. However, the implementation does not perform this verification.

  Reason: The constraints imposed by the requester are either applicable to the QoS features or to the input/output parameters. Since, in the forward expansion stage, the values of the input/output parameters are still unknown and constraint adjustment is part of later stages, their verification is not possible here. Although verification of QoS features is possible during forward expansion, obtaining an optimum quality composition solution is not part of the current goal. Therefore, no verification has been performed in the forward expansion process.
  The correct location for such a verification, if required in future, has been marked in the code. For now, we assume that quality constraints (if any) are always met.

- In the add service algorithm, all the inputs of a new service are placed in a newIn set. While checking if an existing service in the search graph can be a predecessor to this new service, the parameters in newIn are matched with the outputs of the existing service. The parameters that match are then removed from newIn. However, such a removal is not done in the implementation.

  Reason: The algorithm approach restricts some of the possible alternative solutions to the problem. For example, consider two services w1 and w2 that produce the same output

parameters o1 and o2. If a new service w3 takes o1 and o2 as inputs, both w1 and w2 should be allowed as predecessors to w3. This creates two alternative solution branches in the search graph. However, with the algorithm approach, after adding w1 as a predecessor to w3, o1 and o2 are removed from newIn and w2 is never added to its predecessor set. In the implementation, since o1 and o2 are never removed from newIn, w1 and w2 both serve as predecessors to w3.

Future Work

- Various techniques, such as genetic algorithms, can be introduced during search graph construction for improving the quality of composition solutions obtained later.

- A user interface can be created through which the user can apply checks/restrictions on the search graph construction process. For example, the forward expansion process could be stopped once a certain number of possible solutions are obtained, a certain number of layers are constructed, or a certain amount of time has been consumed.