Dantas et al. Journal of the Brazilian Computer Society (2016) 22:6
DOI 10.1186/s13173-016-0047-2

Journal of the
Brazilian Computer Society

RESEARCH Open Access

Assessing the effectiveness of automated service composition

Ramide Dantas[1], Carlos Kamienski[2*], Stenio Fernandes[3] and Djamel Sadok[3]

Abstract

Background: Service Composition is an important feature of Service Oriented Computing, yet it remains mostly a manual process. Given the dynamic and decentralized nature of services, manual composition is a complex undertaking. Proposals to automate this process exist, but suffer from practical problems that hinder their implementation.

Methods: In this paper, we introduce a pragmatic approach where we reverse engineer a service composition repository to obtain the necessary information for automated solutions to work. We then evaluate the quality of the automated compositions based on their similarity to the ones written manually. A classic planning algorithm was adapted in order to generate solutions closer to those expected by developers.

Results: The use of classical planning tools is too time-consuming for agile development scenarios. A simplified, tailored implementation can be orders of magnitude faster than a generic planner, which suggests that expressive power may need to be sacrificed in favor of usability. Our evaluation showed that ensuring the adherence of the solution to the initial specification by enforcing the use of all input parameters was capable of significantly increasing the quality of the solutions.

Conclusions: It is possible to increase the quality of automated composition by applying planning algorithms specially crafted for the service composition task. Comparisons with automated planning tools highlight the effectiveness of our proposal.

Keywords: Automated service composition, AI planning, Performance evaluation, Web services composition

Background

The Internet is being used by various organizations to provide and consume services in order to leverage their businesses and explore new opportunities. Amazon, for example, offers the Simple Workflow Service [1], which allows organizations to host and maintain businesses processes and services using Amazon's cloud infrastructure. With services varying from access to stock prices and weather forecasts to advanced enterprise-wide services, the Internet has become a playground for innovative and ambitious applications developers. Those able to take advantage of this ecosystem to create new applications quickly stand better chances to leap ahead of competitors.

The OASIS organization defines service as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [2]. New services can be built from more general ones, which might be composed to provide new functionalities. This task, however, is highly complex and time consuming. The increasing number of services imposes a challenge to service developers, and the lack of central organization of these services adds to the problem.

The difficulty in composing new services out of existing ones has led researchers to pursue new ways for a smoother service development process. The goal is to streamline service creation in order to keep pace with the dynamicity of technology markets. The best scenario for efficient service creation would be to automate completely the task of composing services, although that

* Correspondence: cak@ufabc.edu.br
[2]Federal University of ABC (UFABC), Av. dos Estados, 5001, Santo Andre ZIP: 09210-580, SP, Brazil
Full list of author information is available at the end of the article

brings new challenges by itself. Several approaches have been proposed to address the automation of service composition. They try to automate the process of finding and combining services to achieve a user-specified functionality [3].

Artificial intelligence (AI) planning is a recurring technique to approach the problem of automatic composition. Using logic-enabled semantic languages provided by the Semantic Web (SW) [4] community, automated composition tools are able to find service compositions that meet the developer requirements, but only if the underlying services are correctly described in terms of their capabilities. Such approaches go beyond the typical Web service description by looking at enriched input and output descriptions [5–7], and pre- and postconditions [8, 9].

There are a few obstacles, however, that prevent the adoption of such proposals in practice. For example, the formal description of services is a manual process, thus, time-consuming and error-prone, which requires new skills from service developers. Even if services are correctly described, the problem of finding a set of combined services that are able to provide certain functionalities is considered a complex undertaking [10, 11]. The existing approaches tackle the problem by limiting the complexity of the solutions, thus reducing the search space, but also the range of possible solutions. Assuming developers find a solution, they may not be willing to immediately deploy it without checking if it meets their expectations. The developer will probably have to tweak with the result of the automated process, by fixing and adapting the composition to ensure that it achieves its purpose. The quality of solution will determine how much extra effort will be put into this task.

In this paper, we approach automated composition as an auxiliary tool for the developers (and not a replacement); we believe that it will be a long way before complete automation of service creation can be achieved. Instead of using manually written semantic descriptions, we use a simple process to extract service compatibility information from existing service compositions, the myExperiment repository [12]. The quality of the automated solutions is estimated by comparing them to handwritten ones in the repository. We found that the quality of solutions using classic AI planners is acceptable, but it can be improved with pertinent modifications to the algorithms, what we showed by adapting a classic AI planning algorithm.

This paper is organized as follows. First, we cover related work, followed by our approach to extract information from existing compositions. In the sequence, we present our modified algorithm, the methodology used to compare compositions and our results. Finally, we bring in-depth discussions on our findings, draw some conclusions, and point out directions for future work.

Background and literature review

This section initially provides some background on the service orientation paradigm and composition. It also presents relevant related work on extracting composition information automatically as well as approaches for automated service composition.

Service composition

A service-oriented architecture (SOA) employs "services" as the basic unit for the separation of concerns in the development of computing systems [13]. Services can be seen as the means whereby consumers access providers' capabilities. Among other features, services provide loosely coupled interaction between partners in a business process (or any other computing activity). Composing services into business processes is a fundamental, yet potentially complex task in service-oriented design and a key feature of SOA. SOA advocates that any service must be capable of participating as an effective composition member.

A service composition is a coordinated aggregation of services that have been assembled to provide the functionality required to automate a specific business task or process [13]. Service orchestration and choreography are two common concepts used to deal with the complexity of service composition [14]. Orchestration, the most common type, refers to coordination of a single process, specifying control and data flows. The most representative language for orchestration is the Business Process Execution Language for Web Services (WS-BPEL) [15]. Service choreography refers to the protocol that ensures harmony and interoperability among the interacting participants (processes), in order for the processes to cooperate with no dependency on a centralized controller.

Service composition or creation techniques can be broadly classified into three categories: (a) Manual service composition occurs when service design is done completely by a human operator, who may or may not use a tool for assisting him/her in generating the composition; b) Automated service composition is based on an intelligent technique, usually AI planning, which automatically finds optimal services for the composition; the ultimate goal in service composition is automation, where from a higher-level service specification the entire process or workflow is generated along with the constituent services and their invocations; (c) Semi-automated service composition lies at some point along the line between the extremes of exclusively manual or automated composition, which may include a myriad of semi-automated techniques, such as the quasi-manual and pattern-based ones proposed in [16]. In this paper, we focus on fully automated service composition.

The most common approach to automated composition is to employ AI-based planning techniques, in order to

find execution plans that, in the end, will make up the service composition. These approaches require the specification of initial and final states in order to derive a path of services that meet these requirements. The services should also provide some kind of semantic description of its effect in the system state, in order for them to work.

Extracting composition information

The analysis in this paper was inspired in the work presented in [17] and [18]. In these research studies, service relations were explored based on their use in real compositions obtained from a repository—myExperiment and ProgrammableWeb, respectively. Their investigation focused on extracting these relations to obtaining useful knowledge about services and composition. One of their findings is that service reuse is low in both cases, i.e., a small number of services are frequently combined together while the majority is used in an isolated way. In their follow-up research, the authors used the service relation graphs to provide to workflow developers GPS-like routing [19] and service recommendations [20, 21]. In this paper, we are interested in finding relations that may be used in automating the creation of complete compositions. We focus on the structure of service and parameter relation instead of looking at other external sources of information about the service (e.g., authorship) since they often are neither available nor reliable. The main differences between our paper and those references are that references [17] and [18] focus on workflow analysis as we do, but not for the purpose if improving automated compositions. On the other hand, references [19, 20] and [21] share some similarities in the overall approach, but aim at service recommendation rather than at fully automated compositions. They also lack a proper comparison with other solutions, whereas we compared our algorithms against planners with the specific goal of assessing the limits of the solutions.

McCandless et al. [22] also used the idea of extracting semantic information—parameter type information, specifically—from WSDL files for the purpose of automated service composition. In that work, however, the authors did not have a repository of compositions to extract additional information on the services relations. McIlraith and Son [23], Traverso and Pistore [24] and Wu et al. [25] reverse engineered OWL-S service descriptions and derived abstract methods telling how to use the services. These works use the service's process model, which provides an abstract description of how clients have to interact with a multi-step service (i.e., a service that can be partially invoked in several steps). Unlike them, we adopt a single step, request-response service model (equivalent to atomic processes in OWL-S and a WSDL operation), and extract information on how various services interact with each other from concrete compositions.

Automated service composition

Several research studies propose automatic composition using only the services' input and output parameters described with semantically enhanced types [22, 26–28]. The new composition is described in terms of what it should receive as input and what result it should render at the end. Services are connected together so that one service output(s) can be used as input(s) to the next service (or any following service) and so on, provided that parameter types match. In such an approach, the more meaning the parameters types carry, the more precise will be the composition found. On the other hand, a composition process based only on primitive types can combine completely unrelated services.

Service composition using preconditions and effects—besides input and output parameters—is a very powerful approach that can theoretically find correct-by-construction execution plans that meet the developer's requirements. Such approach—used by Agarwal et al. [8], Akkiraju et al. [9], Klusch and Gerber [29], McIlraith and Son [23], Rodríguez-Mier et al. [30], Sohrabi et al. [31], Traverso and Pistore [24], Wu et al. [25], among others—has a few practical obstacles in order to be put into production scenarios. There is the need for complete formal descriptions of each service, sometimes requiring a detailed description of service interactions, as in [23, 24], for example. These descriptions must be as detailed and as correct as possible in order to ensure that sound compositions are generated at the end. Finally, the ability to write descriptions using logic formalisms is not a typical skill of service developers, more accustomed to object-oriented languages and graphical IDEs.

Other approaches take a more pragmatic view of automated composition, using it as an auxiliary tool to the developers. In [13, 21] and [32], service recommendations are provided to the composition developer based on various parameters, such as the historical usage pattern of the services, as in [21] or according to the services the developer has already included in the composition, as in [32]. In [19], an analogy is made with the GPS guiding system, where service "routes" are suggested to the developer as he/she builds the composition. In both [19] and [21], the myExperiment repository was used to gather information on how to combine services, as we did in this work, but with focus on service recommendations. Also for scientific workflows, [33] proposes a recommendation service that uses sequence mining, a data mining technique for finding frequent sequential events in a dataset. Our paper, on the other hand, focuses on the challenge of automated service composition based on typical relationships between services, such as the compatibility between input and output

parameters. Also, unlike those papers, our approach is based on AI planning techniques.

In this work, we use a simple distance metric to calculate composition similarity, which is a rough approximation of the number of edit changes the developer would make to the resulting composition. Another measure of this edit distance between compositions is given in [34], where structural features are considered. In their case, the measure was used for matching process descriptions of services to user queries.

Composition quality is seen as a function of the matching of services' input/output parameters in [26] and [35]. Lécué and Mehandjiev [26] calculated the degree of compatibility between parameters based on the semantic matching (Exact, Plugin, Subsume, etc.), thus requiring semantic descriptions. Skoutas et al. [35] investigate dominance relations between matching of parameters, introducing the notion of uncertain (probabilistic) dominance to cope with cases in which it is not possible to establish a clear dominance between matches. For simplicity, in our approach, we consider zero-one compatibility between parameters based on historical data, although we gather information (e.g., number of times these parameters were combined) that could be used for computing composition quality.

Research design and methodology

In this section, we introduce the dataset from the MyExperiment scientific workflow repository and our algorithm for automated composition based on the Graphplan planning algorithm.

Analyzing the MyExperiment repository

Semantic annotations of services are a key part of the automated composition process since it allows algorithms to decide whether a service is appropriate for a specific task or not. However, these semantic descriptions are not commonly found in services, which provide only syntactic descriptions of their interfaces (i.e., WSDL). In order to address this problem, we gather semantic information from existing services based on their usage patterns. Our process consists of analyzing a repository of existing service compositions—the myExperiment project [12]—from which the relations among services can be extracted to be later used for creating new compositions.

In the myExperiment repository, service compositions are called workflows, and therefore, the terms "composition" and "workflow" are used interchangeably throughout this paper. Workflows are publicly available and can easily be obtained via HTTP. We opted to use only Taverna 2 [36] workflows (various languages are supported by the repository, Taverna versions 1 and 2 being the most commonly used). Control structures inside the workflows were not taken into account. Finally, malformed workflows were excluded from the dataset, leaving 425 "good" workflows in the end. The total number of services gathered from these workflows was 1094, 211 of which were SOAP/WSDL Web services (19 %) and 22 were RESTful Web services (2 %); the majority being internal Taverna accessory services (e.g., data manipulation). Most workflows make 10 or less service invocations (more than 80 %), the more invocations the less frequent in the dataset. Another characteristic is low reuse of Web services: 80 % of them were employed in only one workflow (Taverna services are reused more often though).

Taverna workflows contain activities of various types, not only Web services invocations. In this work, we opted for using all kinds of activities as if they were regular services, not only proper Web services, so that complete composition could be built out of the pieces in the repository. The activity type, input and output parameters, and extra information (depending on the activity type, for example, the WSDL URL for Web services) were used to generate unique identifiers.

Input/output parameter compatibility

The first step to determine whether two services are compatible is to check whether an output parameter of one service can be passed as an input to the other. Our objective was to build a parameter compatibility matrix from the set of manually written compositions. We analyzed the assignments of service outputs into service inputs and added the corresponding compatibility relations to the matrix.

The parameter compatibility matrix contains not only the binary relation ("compatible or not") but also the number of times the input and output parameters were connected in different workflows. This number can be used, for example, to rank the output parameters that match certain input parameters, based on how frequently the connection output-input was used. In our experiments, however, we simply consider if the parameters are compatible or not.

Table 1 shows an example of compatibility matrix for a fictitious banking scenario. In such an example, the parameter of type *GetBalance_Balance* can be assigned to parameters of type *SetBalance_NewBal* since in two

Table 1 Parameter compatibility matrix for a simple banking scenario

Output parameter type	Input parameter type				...
	GetBalance_AccNo	*SetBalance_AccNo*	*SetBalance_newBal*	*GetAccNo_ID*	
GetBalance_Balance			2		
SetBalance_Status					
GetAccNo_AccNo	1	3			
GetCPFbyName_ID				4	
...					

compositions the output *Balance* of service *Get_Balance* was connected to the input parameter *NewBal* of service *SetBalance*. The same reasoning follows for the remaining cases in the matrix; blank cells—i.e., assignments that did not occur in the compositions—are assumed to represent incompatible parameters. This is a simplification since the absence of use in compositions does not imply incompatibility necessarily, but allowing full compatibility between parameters would render composition times impractical and generate poor results. Further work is required to evaluate the feasibility of non-binary parameter compatibility.

After analyzing the set of compositions, we extracted 1568 input and 1422 output parameters. The resulting compatibility matrix is very sparse, with only 0.06 % of the possible output-input assignments present. An input parameter is assigned in average 1.6 times, while output parameters are used 1.5 times on average.

Service precedence

A relation explored with the purpose of improving the composition process was the service precedence relation. This relation, of the form "S_1 precedes S_2," happens when service S_1 appears in the execution sequence of the composition before service S_2. Taverna workflows specify data flows instead of control flows, hence not carrying service ordering explicitly. Service precedence is then defined in terms of the dependency of input parameters. In other words, S_1 precedes S_2 if, for any given composition, any input of S_2 depends on an output of S_1 or S_1 precedes any of the services S_2 depends on.

The overall pattern of the workflow-based relation is present in this graph, with a component carrying 65 % of the nodes and several smaller components present. Compared to the previous relation, service precedence presents a larger network diameter and a slightly lower cluster coefficient.

Another interesting relation is the one in which services have direct dependencies within compositions, i.e., services whose input and output parameters are directly connected. The service dependency relation is the more restrictive one in terms of the service connections present compared to the two previous cases. Please recall that, in this graph, two services are connected only if one service uses the output of the other service in any composition. This graph helps to illustrate how often services are directly connected. The average degree in this case is close to 1, i.e., in general, the output of a service is useful to a single other service only.

Adapting a composition algorithm

The problem of composing services is very similar to the problem of finding a suitable plan to execute a task: starting from some given initial state (the known inputs), find a sequence of actions (services) that achieve the desired goal (provide the expected outputs). This perfect match explains why automated AI planning is the preferred technique for addressing the problem of automatically composing services, and it was the path chosen here. In this work, we use classical planning with extensions as well, where the generated plan corresponds to the service composition found.

We assume services are applicable once the information they need is available, that is, there exist some data that match their input parameters, usually the output of another service. Therefore, the state of the system in a given moment is determined by the parameter instances (data) available, and the precondition of a service is its set of input parameters. Parameter instances are consumed but cannot disappear, i.e., there are no negative effects on invoking a service (which only adds more data to the system). Therefore, no mutual exclusion relations will exist between services in our approach (a service cannot prevent another from being called by suppressing its inputs), making the problem easier to solve. In our case, additional constraints are added in order to improve the quality of the solutions, as discussed in the following sections.

In this paper, we use the Graphplan (GP) planning algorithm [37], known as being fast and fitting well the properties of the service composition problem. While exploring alternatives, we also adapted and tested the Fast Forward (FF) algorithm [38]. The results showed FF as being faster than GP but slightly less reliable (i.e., failing more often to find a composition). Given the purpose of this paper of assessing composition in general and the simplicity of the GP algorithm, we focused the remaining of the paper on it.

Graphplan works by building a planning graph of a relaxed version of the planning problem and then attempting to extract a valid plan from the planning graph. If no valid plan is found, the planning graph is expanded further and the process is repeated until a valid solution is found. In the Service Composition analogy, the layers are comprised of service invocations and the parameter instances associated to them.

The expansion of the planning graph is polynomial both in time and in space with the size of the planning problem (number of actions and propositions involved). The most computationally demanding part of the algorithm is the plan extraction, which searches the state space provided by the planning graph for a valid plan.

The Graphplan algorithm suits well the Service Composition problem since the problem, in its basic form, is already "relaxed"; because services have no negative effects, no mutual exclusion happens between them. One consequence is that the search algorithm will not fail to build a plan once the expansion has reached a layer in

which all intended outputs are present. The relaxed version of Graphplan becomes polynomial to the total number of actions, or services in our case [37]. We also made some changes to the algorithm in order to obtain better solutions, as will be discussed in the following sections. These changes, however, reclaim the original complexity of Graphplan.

Enforcing input parameters

General-purpose planners try to find plans with the minimum number of actions. Similarly, the Graphplan algorithm finds plans with minimum execution depth, i.e., the solution found should have the fewest layers possible. It is reasonable to assume that a faster plan is preferred in the general case (i.e., a plan with fewer levels). However, sometimes this leads to plans that do not make use of all information provided in the initial state. If we have services in the service repository that need no inputs, it is not uncommon for the planners to generate compositions that use none or just a few of the inputs provided. Although correct, these compositions may not be exactly what the developer intended, i.e., they may not contain the services the developer would expect to find given his/her specification.

In order to offer more control to the developer, we propose an extra configuration option to the composition specification: the maximum number of unused inputs, $\Delta_{max} \geq 0$. This number tells how many of the provided input parameters can be left unused by the resulting composition. If the number is 0 (zero), then all inputs must be present in the final solution; if it equals or is greater than the number of inputs, then we have the original behavior of the (relaxed) Graphplan algorithm.

We implemented this feature by changing the search procedure of the Graphplan algorithm to account for the number of used inputs. If the search procedure reaches the top layer, it checks the used inputs and, if it meets the specified configuration, it then returns with success; otherwise it backtracks and another search round is made with another set of services. The general behavior of the algorithm follows that of Graphplan with mutual exclusion.

Figure 1 presents the modified GP extraction algorithm. Lines 8 through the end contain the basic Graphplan plan extraction algorithm, where a plan maps to a composition in our scenario. The graph G results from the expasion phase of the algorithm, and contains k levels of actions (services) and propositions (parameters, both inputs and outputs of such services), the top level being the initial set of propositions s_0 (in our case, the set of input parameters for our composition). The goal of the extraction procedure is to reach the top level ($k = 0$) while avoiding conflicting sets of actions. The sets of mutually exclusive propositions (parameters), which configure conflicting actions, are kept in the *nogood*(k) set for each level k. As explained earlier, however, since services are not mutually exclusive, this part of the algorithm it not applicable.

```
   Input: Graph G, goal propositions g and layer k
 1 if k = 0 then
 2 |  if |s_0 \ g| ≤ Δ_max then
 3 |  |  return empty plan // Success
 4 |  else
 5 |  |  return FAIL
 6 |  end
 7 end
 8 A = all sets of actions in layer k of G that provide g
 9 if A = ∅ then return FAIL
10 foreach set of actions in A do
11 |  g' =preconditions(actions)
12 |  if g' ∉ nogood(k) then
13 |  |  plan =extract(G, g', k − 1)
14 |  |  if success then
15 |  |  |  Add actions to level k of plan
16 |  |  |  return plan
17 |  |  else
18 |  |  |  Add g' to nogood(k)
19 |  |  end
20 |  end
21 end
22 return FAIL
```

Fig. 1 Plan extraction algorithm (extract) with unused inputs verification

When first the extract procedure is first invoked, the set g contains the final output parameters of the composition. In the next invocation of extract(), this set becomes the inputs passed to the services that generate the final outputs. This process is repeated for each recursive call as the search continues upwards. At the top level ($k = 0$), one should expect g to match the original set of initial input parameters s_0 provided by the user, but this is not necessarily the case. To enforce this, we modified the procedure as seem in line 2, where it is verified if the current set of parameters g differs in at most Δ_{max} from the set of original input parameters s_0. If this condition holds, the solution is accepted; otherwise, a failure is returned and the extraction continues searching other paths.

The feature of enforcing the use of input parameters aims at improving the adherence of the final solution to the specification and find solutions closer to the ones a human developer would build. One can see it as a special case of Planning with Preferences [39].

Preferred services with reverse Graphplan

In order to improve the performance of the solution extraction algorithm, whenever it has to select services, we use a relevance pruning technique we call Preferred

Services, which is similar to the approach presented in [40]. In order to compute the preferred services set, we apply the relaxed Graphplan expansion algorithm both in the regular direction (from the inputs to the outputs) and in the reverse direction (from outputs to inputs). The resulting planning graph for each phase is "reduced," a process that removes unnecessary services. The set of preferred services is given by the union of the services comprising both the direct and the reverse reduced planning graphs.

Graphplan in its original form does not guarantee that all inputs provided will be part of the final solution, although the solution must provide all outputs. Conversely, applying the Graphplan expansion in the reverse direction, we have a planning graph where all inputs are reachable but not necessarily all outputs are used. The intuition is that combining the services in these graphs would provide the set of services likely to be part of the final solution. Before merging the graphs' services, though, we remove from them the services that are not in the path between an input and an output. This reduction algorithm removes the invocations at the last level whose outputs (or inputs, in the reverse case) are not outputs of the composition (inputs, respectively). It proceeds upwards, removing invocations, whose outputs are not used in the level below, until it reaches the top level (Fig. 2).

The set of preferred services is used in our Graphplan variant during the solution extraction phase, which, in hard cases, is the more time-consuming phase of the algorithm (as for the original Graphplan algorithm). In this phase, the algorithm has to choose between invocations at a given level that provide the same input to an invocation in the level below. The algorithm then uses the set of preferred services to guide this choice, ranking the candidate invocations depending on whether the associated service is preferred or not.

Methods

We assume that the service developer will have to deal with the output of such automated composition tools. Therefore, the usefulness of these tools will depend on how much effort can be saved by applying them. Instead of directly assessing the effort gain or loss, we approximate it by measuring how close to handwritten compositions are the automated solutions found by the algorithms, what we call the quality of the compositions. Since we have the original compositions from the myExperiment repository at hand, we are able to compare the output of the algorithms with the original ones.

This evaluation consists of rebuilding each original composition from the myExperiment repository by submitting its interface description (i.e., input and output parameters) to the algorithms and measuring the quality of the solution found. The more similar the solutions to the original compositions, the less effort the developer would have to spend adapting and fixing them.

Composition quality

The quality of a composition can be a very subjective matter. Here, the quality of a solution is a measure of how similar to a handwritten composition it is for the same initial specification. The similarity between compositions can be measured with different levels of detail. Instead of a very precise similarity measure—e.g., considering individual connections between services and the order of services inside of the compositions—we opted for a high-level similarity metric that uses only the list of services present in both compositions, irrespective of the order or number of occurrences of these services within the compositions. The premise is that finding the correct services for a given composition represents a fundamental part of creating a new composition manually. If an automated process could generate compositions with the correct services, even if not connected in the way the developer would expect, this could save one a considerable amount of time. We assume that in a real scenario, a human developer will validate the compositions found by our algorithm, as we already have an experience with a preliminary version of it used in a service composition tool for the cloud [41].

Input: Graph G, goal propositions g and layer k
Output: Reduced graph G

```
while k > 0 do
    foreach action in level k of G do
        if all effects of action not used at level k + 1 or not in g then
            remove action from level k
        end
    end
    k = k − 1
end
```

Fig. 2 Planning graph reduction algorithm

We used a Jaccard similarity coefficient: the percentage of services correctly found by the composition process with respect to the total services on both compositions (automatic and manual). We chose Jaccard mostly due to its simplicity. Also, it fits well our purpose, by telling the similarity between the original set of services and the one computed by our algorithm. Other similarity metrics are available, such as the Dice coefficient, which is more useful for some string similarity applications. Both Jaccard and Dice compare sets or strings but the former consider exact matches, whereas the latter considers bigrams. Also, there are other metrics but they are mostly variations or generalizations of Jaccard or Dice, which make them more complicated and not necessarily fit to our purposes.

Formally, the similarity between two compositions C_1 and C_2 is given by the formula (1):

$$\text{similarity}(C_1, C_2) = \frac{|\text{services}_{C1} \cap \text{services}_{C2}|}{|\text{services}_{C1} \cup \text{services}_{C2}|} \quad (1)$$

A similarity($C_{\text{orig}}, C_{\text{manual}}$) = 0 means that the automatic (C_{auto}) and manual (C_{manual}) compositions do not share any services, while *similarity*($C_{\text{orig}}, C_{\text{manual}}$) = 1 means that both compositions share the same set of services. Two closely related metrics are the hit rate (2), which reflects the number of correct services ("hits") found, and the error rate (3), that measures the number of false positives, i.e., services the developer will likely remove from the final composition.

$$hit(C_{\text{auto}}, C_{\text{manual}}) = \frac{|\text{services}_{C\text{auto}} \cap \text{services}_{C\text{manual}}|}{|\text{services}_{C\text{manual}}|} \quad (2)$$

$$err(C_{\text{auto}}, C_{\text{manual}}) = \frac{|\text{services}_{C\text{auto}} - \text{services}_{C\text{manual}}|}{|\text{services}_{C\text{auto}}|} \quad (3)$$

Baseline planners

In the experimental evaluation, we used a state-of-the-art AI planner as well as the algorithm described in previous section. We selected Fast Downward (FD) [42] planner because of its performance in the last International Planning Competition [43]. FD is a forward planner, i.e., it searches the state space from the initial state until a goal state is found. Instead of using the planning graph as a heuristic as in the Fast Forward algorithm, the FD planner adopts another structure called "causal graph" as the base structure for computing heuristic values. The causal graph is defined by causal dependency relations between objects of the planning domain: two objects have a causal dependency if the state of the first object is changed by an operation whose precondition depends on the state of the second. We used three FD heuristics (or variations) from the IPC2011: FDSS1 [44], SelMax [45] and LAMA2011 [46]. FD uses PDDL as input language for the planning problems.

We had to reduce the set of compositions from 425 down to 300 elements in order for the FD planner and algorithm to run within our resources; our algorithms were able to run with the complete dataset within the same constraints. Additionally, we excluded some compositions from the 300-workflow dataset to create a more realistic scenario for the algorithms and prevent distortions in the results: compositions with no output parameters (not suitable for the planners), compositions containing one single service, and "isolated" compositions, whose services are not used by any other composition in the repository.

Each individual composition attempt was allowed to use up to 8GB of memory and to run for 30 min, which is the time limit used in IPC competitions. We implemented our algorithms in Java and executed with the same time and memory constraints. The basic Graphplan algorithm from the literature, adapted for service composition, is identified in the evaluation as *GP*. Our *GP* variant that supports enforced inputs is identified by GP_{enf} ($GP_{\text{enf}}^{\text{pref}}$ when the preferred services feature is used).

Results and discussion

Evaluation results are presented in this section, according to the methodology and methods introduced in the previous sections. Selected results cover the performance of the planning tools, the quality of the solutions and the use of input parameters and service precedence. Also, we discuss our results in the end of the section.

Performance of planning tools

The first scenario we evaluated aims at measuring the performance of standard planning tools with respect to both quality and speed. We compared the FD variants—*FDSS*1, *SelMax* and *Lama*2011—to our standard Graphplan implementation (*GP*), using the 300-composition dataset. Table 2 summarizes the results for these first experiments, showing the number of cases in which each planner succeeded, failed, or terminated due to timeout, along with the average time per experiment (plus/minus the standard deviation).

Table 2 Evaluation summary for the main scenario

	*FDSS*1	*SelMax*	*Lama*2011	*GP*
Success	199	122	194	212
Failed	3	1	4	0
Timeout	10	89	14	0
Time (avg ± sd)	519 ± 282 s	383 ± 59 s	516 ± 243 s	30 ± 65 ms

In this scenario, we aim at rebuilding compositions using information extracted from them in the first place; therefore, we should expect the tools to be able to find solutions to all experiments. The first notable observation is that only the *GP* algorithm was capable of that; the *FD*-based heuristics failed for some compositions due to timeout or were not able to find a solution at all (before timeout). Among the *FD* heuristics, the *SelMax* had the worst success rate, failing in more than 40 % of the cases (including timeout cases); its companion heuristics had failure rates below 10 %. Time figures in Table 2 are only illustrative of the computational effort required and should not be used as precise estimated of the composition time in real scenario. General purpose planners—-such as FD—provide features not used in this evaluation and are highly optimized for solution quality instead of computational speed. They also suffer with PDDL-parsing overhead, unlike our GP implementation.

Quality of the solutions

The quality of the solutions, measured by the similarity metrics is presented in Table 3. Overall, the quality of the solutions generated by the *FD* heuristics and the *GP* algorithm are around 75 %, with the exception of the *SelMax* heuristic. This heuristic, however, has the worst performance in terms of success rate, being able to solve only 60 % of the cases tested.

As previously mentioned, general-purpose planners focus on finding the shortest plan (in number of services or time steps); using all the inputs provided is not a priority. To illustrate such a fact, Fig. 3 presents the composition found by both the *FDSS*1 and *GP* planners for a given workflow in the myExperiment repository. The solution for this case does not use all the original input parameters—NumRunYear is left unused. As a result, it lacks one of the Web services of the original solutions—*wsdl* : RunDynamicSimple – and the solution quality is 0.67.

Using input parameters

We addressed this problem with extensions that enforce the use of the inputs provided by the developer. We tested these algorithms against the *FDSS*1 heuristic. In Table 4, we present the results for the enforced *GP*, with and without the use of Preferred Services (GP_{enf} and $GP_{\text{enf}}^{\text{pref}}$ respectively); the results for *FDSS*1 and basic *GP* are repeated for easy reference.

Table 3 Composition metrics—main scenario

	*FDSS*1	*SelMax*	*Lama*2011	*GP*
Similarity	0.77	0.88	0.75	0.73
Hit rate	0.79	0.90	0.77	0.78
Error rate	0.08	0.04	0.10	0.14
Services (avg)	4.14	4.18	4.07	4.74
Depth (avg)	3.24	3.27	3.25	3.33

Observing, only the quality obtained by the algorithms, the improvement seems unimpressive. The modified algorithms matched the quality provided by the *FD* planner, being also able to improve the hit rate—i.e., number of correct services found—from 0.79 to 0.83. The real value of the modified GP algorithms, however, comes out when we look only at the cases in which they made a difference: instances where traditional planning algorithms did not use all the information provided by the developer in the solution. The average quality for these cases rose from 0.45, using the *FDSS*1 heuristic, to 0.71 using the enforced *GP* algorithm with preferred services ($GP_{\text{enf}}^{\text{pref}}$), as can be seen in Table 5. The hit rate also increased considerably: while the *FD* planner found around half the services the developer wanted, our algorithm was able to find as many as 80 % of them.

Using service precedence

Finally, we experimented with the service precedence information extracted from the composition repository. The service precedence relation can be used to determine if a service *X* is a prerequisite for service *Y*, even if they are not directly connected. Intuitively, if we observe that service *X* precedes service *Y* in all compositions where service *Y* is used, then one can assume that *X* is needed in order to invoke *Y*, i.e., *X* is an implicit precondition of *Y*. For this evaluation, we made the precedence relation an explicit precondition of the services in our repository. No modifications were made to the algorithms, since they are able to handle these simple preconditions natively. We evaluated the main scenario with precedence information using the *FDSS*1 planner and the basic *GP* algorithm; the results are shown in Table 6.

The use of the additional information had an impact on the quality of the solutions in both cases. Graphplan outperformed *FDSS*1 with respect to the hit rate metric (0.86 against 0.83) but suffered from the large number of services in its solutions, which reduced its overall similarity. In this evaluation, service precedence was used as a hard precondition: a service is only added to a solution if all services it requires appear before its use in the composition. In the future, we plan to use the precedence information in a softer way, deferring instead of pruning away the services that do not have all of their required services fulfilled.

Discussion

Automated Service Composition has already been studied for a number of years and several proposals have

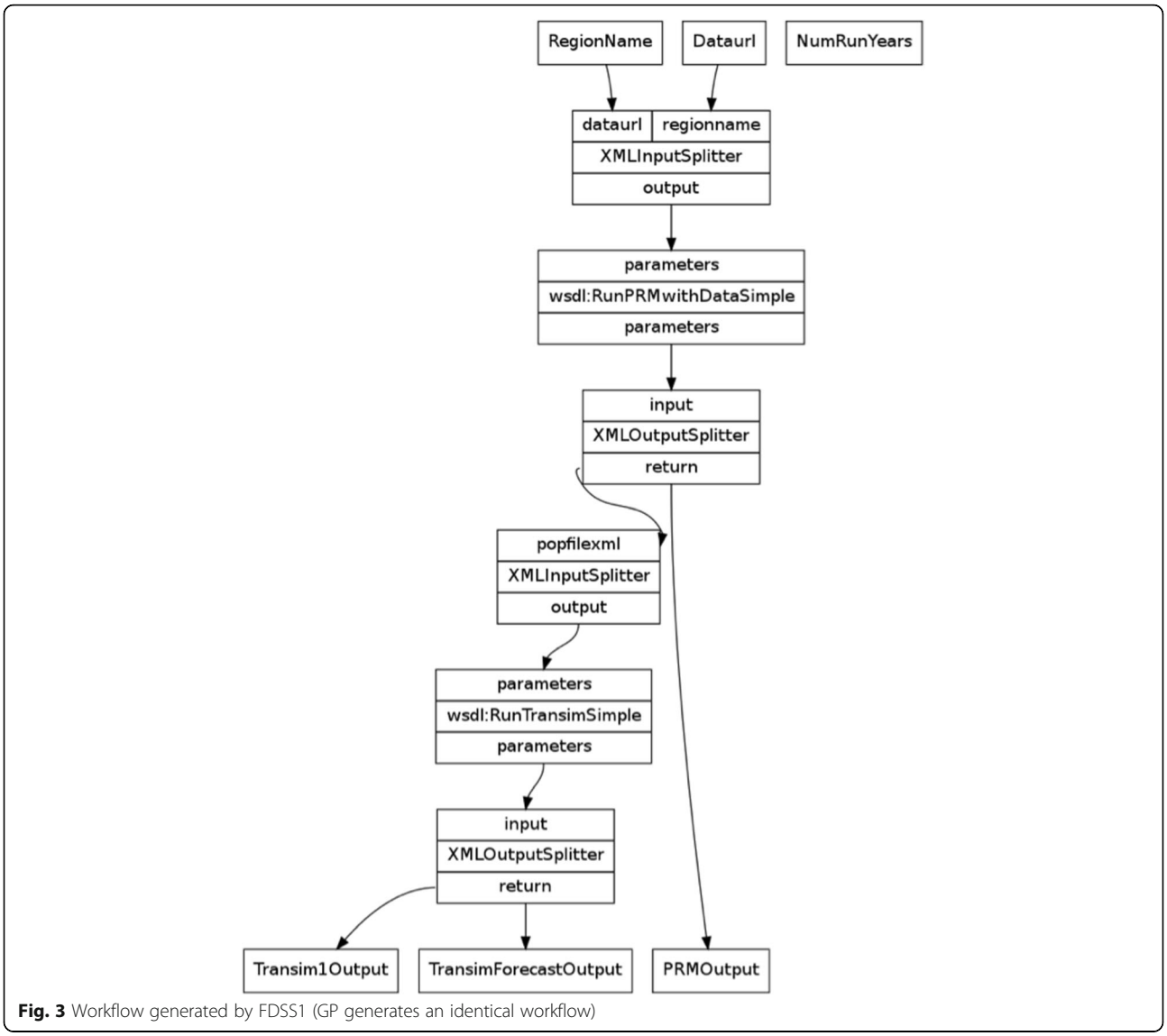

Fig. 3 Workflow generated by FDSS1 (GP generates an identical workflow)

emerged with varied levels of maturity. Nonetheless, the use of automated composition techniques by service developers has not taken off yet, despite the efforts of the Web services community. In our opinion, this is due, in part, to the fact that some complex approaches seek to reduce the role of the developer in the (automated) composition process, while even trying to replace him/her altogether; other simpler approaches, conversely, may be too dependent on the developer, requiring constant feedback in order to find solutions. We think that automated composition can be an important item in the developer's utility belt—akin to code completion and refactoring in modern integrated development environments—but for that to happen it must be simple to use,

Table 4 Composition metrics enforcing the use of input parameters

	*FDSS*1	*GP*	GP_{enf}	GP_{enf}^{pref}
Similarity	0.77	0.73	0.78	0.77
Hit rate	0.79	0.78	0.83	0.83
Error rate	0.08	0.14	0.12	0.12
Services (avg)	4.14	4.74	4.92	5.01
Depth (avg)	3.24	3.33	3.44	3.49

Table 5 Composition quality enforcing the use of input parameters

	*FDSS*1	GP_{enf}^{pref}
Similarity	0.45	0.71
Hit rate	0.54	0.82
Error rate	0.31	0.18
Services (avg)	3.77	5.20
Depth (avg)	3.03	3.91

Table 6 Composition metrics for service precedence

	Without service precedence		With service precedence	
	*FDSS*1	*GP*	*FDSS*1	*GP*
Similarity	0.77	0.73	0.81	0.77
Hit rate	0.79	0.78	0.83	0.86
Error rate	0.08	0.14	0.09	0.14
Services (avg)	4.14	4.74	4.13	5.76
Depth (avg)	3.24	3.33	3.35	3.53

non-intrusive, relatively fast, and more robust to human failures. Moreover, the developer should be able to rely on the solutions provided by the algorithms or at least have some expectation of how reliable (or not) the compositions generated automatically are.

In this work, we have focused on some of the problems that hinder the applicability of automated composition in practical scenarios. To circumvent the lack of semantic information that is common for real-life services, we opted for using a publicly available repository of scientific workflows, from which we extracted the information necessary for the planning algorithms. We measured the accuracy of the automated compositions compared to ones written manually in order to estimate how helpful these tools would be to composition developers. The results showed that the overall quality of compositions obtained by both standard planning tools and our algorithms is acceptable (around or above 75 %), but there was room for improvement for specific cases. We identified that the cases in which not all input information was used in the solutions had lower average quality, and that, by applying an algorithm designed to address this problem, the quality was increased from 45 to 71 %.

The results also showed that a tailored implementation is able to perform faster than a more general one by orders of magnitude, while providing similar quality. This finding, however, comes with the caveat that the PDDL domain description may affect execution times. In our experiments we used a PDDL mapping supported by the all selected planners, although other planners support simpler ways of describing the domain. PDDL processing itself has a cost in the overall performance of the planners as well.

Another interesting result was that using more information associated to the services (i.e., service precedence) could improve the solutions (comparing with plain GP) at the expense of significantly increasing the time required to compute them. In our experiments with and without service precedence, the average time (and standard deviation) was 29 ms (65 ms) and 388 ms (1006 ms), respectively. Service precedence also improved hit rate compared to GP with enforced inputs: 0.86 against 0.83.

The decision of what composition tool to use—a generic planner or our tailored algorithms—varies according to the requirements of the composition developer. If the developer has little or no constraint on the time required to compute the solutions, using a generic planner such as the Fast Downward along with precedence information is a good option since it generates compositions with the best quality on average (but takes several minutes doing so). However, if response time is an issue, as in an iterative development approach, the results show that our algorithms, especially the Graphplan with Enforced Inputs and Preferred Services, provide the best balance between composition quality and computation time.

There are other planning strategies that can mimic the features implemented in our algorithms, namely, planning with preferences (enforced inputs). For the evaluation in this work, however, we were able to compare our algorithms to classical planning tools only, and even then, the evaluation encountered several obstacles due to the inability of the tools to handle large domains. However, as a thorough comparison could not be carried out, we cannot rule them out right away. This comparison is a subject for future work.

Conclusions

Automated composition using standard planning algorithms can indeed provide solutions with encouraging quality levels. It is possible, however, to increase the quality by applying planning algorithms specially crafted for the service composition task. In particular, ensuring the adherence of the solution to the initial specification by enforcing the use of all input parameters showed that it was capable of increasing the quality of the solutions. We measured the quality of the solutions and overall performance of the planning tools and our algorithm. A real-life composition repository was used in the evaluation, generating a planning domain large enough to cause several state-of-the-art planners to crash.

The evaluation also showed that using classical planning tools is too time-consuming for agile development scenarios. A simplified, tailored implementation can be orders of magnitude faster than a generic planner, which suggests that expressive power may need to be sacrificed in favor of usability. Irrespective of the tool, some problem instances will still require more time than the developer may be willing to wait, as timed-out cases happened to most algorithms. Our opinion—shared by other authors [21]—is that the algorithms either find a solution "quickly" or hang indefinitely looking for it.

In this work, we focused on simpler, dataflow-oriented compositions where the main elements are services and the connections between them. Current composition languages, such as WS-BPEL, allow for more elaborate

compositions, resembling traditional programming languages. In our opinion, however, these control structures would require much richer composition specifications than the basic input/output approach. The richer the specification, the more complex will be the developer's job of describing the composition in the first place, to the point where specifying the composition could eventually take the same effort as writing the composition itself. With that in mind, and for the purpose of having automated composition assist—not replace—the developer, having full-blown control-structures in solutions might be overkill. We want to investigate this subject further and verify to what extent these insights are valid or not.

Authors' contributions

RD developed the algorithms and experiments as the results of his PhD dissertation, as well as wrote the first version of the paper. CK worked as RD's co-supervisor, providing valuable ideas, insights, and discussions related to the work and also revised the paper. SF contributed with valuable discussions and ideas and also revised the paper. DS worked as RD's supervisor, contributing in different manners during the whole development of the research, including this article. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Federal Institute of Pernambuco (IFPE), Av. Prof. Luiz Freire, 500, Cidade Universitária, Recife ZIP: 50740-540, PE, Brazil. [2]Federal University of ABC (UFABC), Av. dos Estados, 5001, Santo Andre ZIP: 09210-580, SP, Brazil. [3]Federal University of Pernambuco (UFPE), Av. Prof. Moraes Rego, 1235—Cidade Universitária, Recife ZIP: 50670-901, PE, Brazil.

Received: 11 November 2015 Accepted: 27 September 2016
Published online: 12 October 2016

References

1. Amazon Simple Workflow Service (2015), http://www.aws.amazon.com/swf
2. OASIS (2006) Reference Model for Service Oriented Architecture 1.0, OASIS
3. Dustdar S, Wolfgang S (2005) A survey on web services composition. Int J Web Grid Serv 1(1):1–30
4. Berners-Lee T, Hendler J, Lassila O (2001) The Semantic Web. Sci Am 284(1):29–37
5. Arpinar IB, Aleman-Meza B, Zhang R, Maduko A (2004) Ontology-driven web services composition platform, IEEE Intl. Conference on E-Commerce Technology., pp 146–152
6. Lécué F, Leger A (2006) Semantic web service composition through a matchmaking of domain, European Conference on Web Services., pp 171–180
7. Zhang R, Arpinar IB, Aleman-Meza B (2003) Automatic composition of semantic web services, International Conference on Web Services (ICWS)., pp 38–41
8. Agarwal V, Chafle G, Dasgupta K, Karnik N, Kumar A, Mittal S, Srivastava B (2005) Synthy: a system for end to end composition of web services. Web Semant 3(4):311–339
9. Akkiraju R, Srivastava B, Ivan A, Goodwin R, Syeda-Mahmood T (2006) SEMAPLAN: combining planning with semantic matching to achieve web service composition, IEEE International Conference on Web Services (ICWS)., pp 37–44
10. Berardi D, Calvanese D, De Giacomo G, Lenzerini M, Mecella M (2003) Automatic composition of e-Services. Lect Notes Comput Sci 2910(1):43–58
11. Hoffmann J, Bertoli P, Pistore M (2007) Web service composition as planning, revisited: in between background theories and initial state uncertainty, National Conference of the American Association for Artificial Intelligence., pp 1013–1018
12. myExperiment (2015), http://www.myexperiment.org
13. Erl T (2007) SOA principles of service design, 1st edn. Prentice Hall, Boston
14. Peltz C (2003) Web services orchestration and choreography. IEEE Comput 36(10):46–52
15. OASIS (2007) Web Services Business Process Execution Language Version 2. 0. OASIS. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
16. Azevedo E, Kamienski C, Dantas R, Ohlman B, Sadok D (2014) A path to automated service creation via semi-automation levels. J Braz Comput Soc 20(2):1–20
17. Tan W, Zhang J, Foster I (2010) Network analysis of scientific workflows: a gateway to reuse. IEEE Comput 43(9):54–61
18. Huang K, Yushun F, Wei T (2012) An empirical study of programmable web: a network analysis on a service-mashup system, IEEE International Conference on Web Services (ICWS)., pp 552–559
19. Tan W, Zhang J, Madduri R, Foster I, de Roure D, Goble C (2011) ServiceMap: providing map and GPS assistance to service composition in bioinformatics, IEEE International Conference on Services Computing (SCC)., pp 632–639
20. Huang K, Fan Y, Tan W, Li X (2013) Service recommendation in an evolving ecosystem: a link prediction approach, IEEE International Conference on Web Services (ICWS)., pp 507–514
21. Zhang J, Tan W, Alexander J, Foster I, Madduri R (2011) Recommend-as-you-go: a novel approach supporting services-oriented scientific workflow reuse, IEEE International Conference on Services Computing (SCC)., pp 48–55
22. McCandless D, Obrst L, Hawthorne S (2009) Dynamic web service assembly using OWL and a theorem prover, IEEE International Conference on Semantic Computing., pp 336–341
23. McIlraith SA, Son T (2002) Adapting Golog for composition of semantic web services, International Conference on Knowledge Representation and Reasoning (KR)., pp 482–493
24. Traverso P, Pistore M (2004) Automated composition of semantic web services into executable processes. Lect Notes Comput Sci 3298(1):380–394
25. Wu D, Parsia B, Sirin E, Hendler J, Nau D (2003) Automating DAML-S web services composition using SHOP2. Lect Notes Comput Sci 2870(1):195–210
26. Lécué F, Mehandjiev N (2011) Seeking quality of web service composition in a semantic dimension. IEEE Trans Knowl Data Eng 23(6):942–959
27. Omer AM, Schill A (2009) Web service composition using input/output dependency matrix, ACM Workshop on Agent-Oriented Software Engineering Challenges for Ubiquitous and Pervasive Computing., pp 21–26
28. Yan Y, Zheng X (2008) A planning graph based algorithm for semantic web service composition, IEEE Conference on E-Commerce Technology., pp 339–342
29. Klusch M, Gerber A (2005) Semantic web service composition planning with OWLS-XPlan, International AAAI Fall Symposium on Agents and the Semantic Web., pp 1–8
30. Rodríguez-Mier P, Mucientes M, Lama M, Couto M (2010) Composition of web services through genetic programming. Evol Intell 3(3):171–186
31. Sohrabi S, Prokoshyna N, McIlraith SA (2006) Web service composition via generic procedures and customizing user preferences. Lect Notes Comput Sci 4273(1):597–611
32. Chen L, Wu J, Jian H, Deng H, Wu Z (2013) Instant recommendation for web services composition. IEEE Trans Serv Comput 7(4):586–598
33. Tosta F, Braganholo V, Murta L et al. (2015) Improving Workflow Design by Mining Reusable Tasks. J Braz Comput Soc 21:16. http://link.springer.com/article/10.1186/s13173-015-0035-y.
34. Grigori D, Corrales JC, Bouzeghoub M, Gater A (2010) Ranking BPEL processes for service discovery. IEEE Trans Serv Comput 3(3):178–192
35. Skoutas D, Sacharidis D, Simitsis A, Sellis T (2010) Ranking and clustering web services using multicriteria dominance relationships. IEEE Trans Serv Comput 3(3):163–177
36. Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock M, Wipat A, Li P (2004) Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17):3045–3054
37. Blum A, Furst M (1997) Fast planning through planning graph analysis. Artif Intell 90(1):281–300
38. Hoffmann J, Nebel B (2001) The FF planning system: fast plan generation through heuristic search. J Artif Intell Res 14(1):253–302
39. Baier JA, McIlraith SA (2009) Planning with preferences. AI Mag 29(4):25–36
40. Pettersson MP (2005) Reversed planning graphs for relevance heuristics in AI planning. Planning, Artificial Intelligence Researcher Symposium on Scheduling and Constraint Satisfaction: From Theory to Practice 117:29–38
41. Kamienski C, Simões R, Azevedo E, Dantas, Ramide D, Dias C, Sadok D, Fernandes S (2014) E2ECloud: composition and execution of end-to-end services in the cloud, IEEE Symposium on Computers and Communications (ISCC 2014)

42. Helmert M (2006) The fast downward planning system. J Artif Intell Res 26(1):191–246
43. International Planning Competition (2014), http://ipc.icaps-conference.org. Accessed 05 Oct 2016.
44. Helmert M, Röger G, Karpas E (2011) Fast Downward Stone Soup, Seventh International Planning Competition (IPC 2011)., pp 38–45
45. Domshlak C, Helmert M, Karpas E, Markovitch S (2011) The SelMax planner: online learning for speeding up optimal planning, Seventh International Planning Competition (IPC 2011)., pp 108-–112
46. Richter S, Westphal M, Helmer M (2011) LAMA 2008 and 2011, Seventh International Planning Competition (IPC 2011)., pp 50–54