Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 112 (2017) 643–652

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Knowledge Based and Intelligent Information and Engineering Systems, KES2017, 6-8 September 2017, Marseille, France

Engineering Polynomial-Time Solutions for Automatic Web Service Composition

Paul Diac[a]

[a]*Faculty of Computer Sicence, Alexandru Ioan Cuza University, General Berthelot, 16, Iasi, 700483, Romania*

Abstract

Web Service Composition (WSC) is the task of creating some new functionality over a repository of independent resources, Web Services in particular. Services are described by their input and output parameters that are matched if they have the same textual name in service definition for the classic, simplified version of WSC. The problem requires finding an ordered list of services such that all input parameters are available starting from the initially user known parameters and revealing all user required parameters. In this paper we propose a proven efficient polynomial-time solution to Automatic WSC combined with a heuristic for shortening the solution length: the number of Web Services in the composition. The algorithm is tested against several benchmarks of tests and is compared with previous solutions that use AI Planning, revealing tremendous improvements. Two benchmarks are well-known in WSC literature but due to lack of high run time variations over their tests, a new benchmark is created with a special designed generator described in the paper. The new tests reveal more meaningful information.

Peer-review under responsibility of KES International

Keywords: automatic web service composition; polynomial time; heuristic; planning; benchmark

1. Introduction

Web Services are present in many modern software architectures and they behave as independent components that generally solve only one precise task. More complex systems can take use of them only in a composed manner, chaining or branching them with respect to the services requirements[1]. Several standardization languages have been implemented to enable this, such as WS-BPEL[2], BPEL for Semantic Web Services (BPEL4SWS)[3], OWL-S[4] and more. Enabling the automation of the process can be a complex task from multiple perspectives, especially in future where the number of publicly available Web Services is expected to have a steady growth. The focus in this paper is on classical Automatic WSC problem where a repository of Web Services and a single user request are known. Each service consists of its input and output parameters and the user request has the same structure: the initially parameters known by the user and the parameters to be found. For the user request we need to find a list of services with the

* Corresponding author. Tel.: +04-075-215-3555;
E-mail address: paul.diac@info.uaic.ro

1877-0509 © 2017 The Authors. Published by Elsevier B.V.
Peer-review under responsibility of KES International
10.1016/j.procs.2017.08.110

property that each service in the list can be called: its input parameters are known from the previous services or from the user request and all user required parameters are learned. The problem's solution is a satisfying composition. Solution length is taken into account, shorter solutions are desirable if multiple exist.

The paper is organized as follows: section 1 contains this introduction and section 2 describes the formal definition of the problem. Section 3 shortly presents the previously planning based solutions for WSC. Section 4 presents our evaluation plan: existing benchmarks and their limitations together with a new test generator that is more relevant to running time variations. The proposed solution in section 5 describes our polynomial-time algorithm for WSC with the required data structures and enhanced with section 6's heuristics to shorten the solution length. Last section 7 discusses the experimental results with the conclusion and possible future work.

2. Automatic WSC problem definition

Automatic Web Service Composition can be abstracted to the problem described below. The problem is defined using a graph-like structure with complex nodes and the restrictions on the node path that represents the solution in Web Service Composition.

Node. A node n is defined by a pair $\langle I,O\rangle$ where I and O are sets of parameters. I is the input parameter set for the node and O is the output parameter set. We will write them as $n.I$ and $n.O$.

Parameter set. The set of all parameters that appear in all nodes as input or output; also called *the universe*. If R is the set of all nodes then it is $\bigcup_{n\in R} n.I \cup n.O$.

Initial node. A special node *Init* that specifies the initially known parameters. *Init.I* should conventionally be $\emptyset$ and $i.O$ the set of initially know parameters. This way the user request input parameters can be represented as a regular node.

Goal node. A special node *Goal* that defines the parameters that need to be found. *Goal.O* should be $\emptyset$ since it produces no information, just specifies that *Goal.I* is the set of desired parameters. Informally it is the node that needs to be reached.

Parameter matching. Let P be a set of parameters and n a node. We say that the set P matches the node n if $n.I \subseteq P$. We further define $P \oplus n = P \cup n.O$ as the union of $n.O$ and P under the constraint of P matching n.

Chained matching. If P is a set of parameters and $\langle n_1, n_2, ...n_k\rangle$ is an ordered list of nodes, we say that $P \oplus n_1 \oplus n_2 ... \oplus n_k$ is a chain of matching nodes over the set P if:

$$n_i.I \subseteq \left(P \cup \left(\bigcup_{j=1}^{i-1} n_j.O\right)\right), \quad \forall \ i = \overline{1..k}$$

In words, a chain of matching nodes is a list of nodes for which the input of each node is included in the union of the output sets of each previous nodes and the initial set of parameters.

Node Composition problem. Given a set of nodes R and two initial and goal nodes *Init* and *Goal* find a matching list nodes $\langle Init, n_1, n_2, ...n_k, Goal\rangle$ with $n_i \in R$, $\forall\ i = \overline{1..k}$.

Clearly, each node can be interpreted as a Web Service so that Node Composition is equivalent to **WSC**. One Web Service can be executed only if we know all its input parameters. Therefore one valid **WSC** is translated into node chain matching.

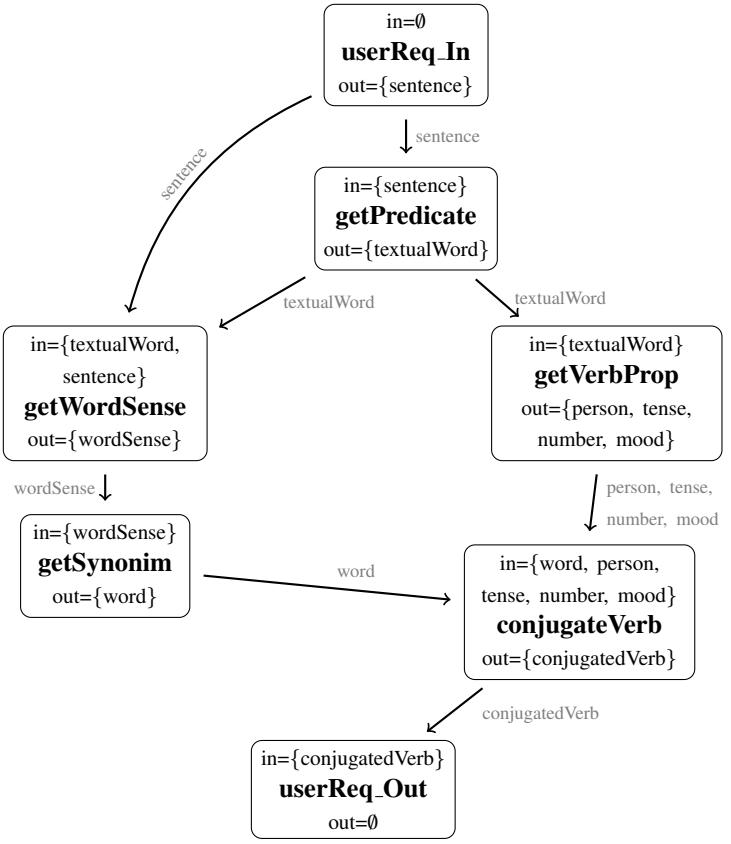
Web Service Composition Example. Suppose that as part of a text processing phase we need to replace the predicate of a sentence with a synonym of the verb that constitutes the predicate. The replacement has to be made in the correct conjugation. However, the service that provides synonyms takes as input not a word but a word sense and and there is also a word sense disambiguation service. More precisely, considering the web services:

$$getWordSense \begin{matrix} in = \{textualWord,\ sentence\} \\ out = \{wordSense\} \end{matrix} \qquad getSynonim \begin{matrix} in = \{wordSense\} \\ out = \{word\} \end{matrix} \qquad getPredicate \begin{matrix} in = \{sentence\} \\ out = \{textualWord\} \end{matrix}$$

$$getVerbProp \begin{matrix} in = \{textualWord\} \\ out = \{person,\ tense,\ number,\ mood\} \end{matrix} \qquad conjugateVerb \begin{matrix} in = \{word,\ person,\ tense,\ number,\ mood\} \\ out = \{conjugatedVerb\} \end{matrix}$$

The user request has the initial parameter *sentence* and wants to obtain *conjugatedVerb*, the synonym of the verb in the correct form, so it can further replace it in the sentence. We need to call the services in such an order that all input parameters are known at the time of a service call, as shown in Figure 1.

Figure 1: *Composition example: arrows show the source of each input parameter of services*



3. Planning-based solutions

Most of existing solutions to Automatic WSC problem use AI Planning, since it is straightforward to reduce WSC instances to planning instances. For example, Zou et al[5] proposes an efficient solution that uses AI Planning, together with a comprehensive analysis of the performance relative to previous solutions. Our solution presented in sections 5 and 6 is compared with this approach over all benchmarks. The reduction of WSC to planning is described in more details below, and is the same as in Zou et al[5] and the one implemented for testing. For solving the transformed planning instances both GraphPlan[6] and Fast-Forward[7] planners are used and compared in terms of running time and solution length. In order to do this, we need to convert between three formats: WSDL (Web Services Description Language)[2], STRIPS[8] used by GraphPlan and PDDL[9] used by Fast-Forward. Shortly, the planning languages use: objects, predicates, initial and goal states and actions. One action is dentified by a *precondition* and *effect*, both of which are described by a first order logic formula. An action can be applied if the *precondition* is *true* and after the *effect* becomes *true*.

The relation with the Planning domain is based on the following reductions: each Web Service is translated into one action in planning problem definition. The service input and output parameters are translated into the *precondition* and *effect* of that action. The only predicate used is the *have* predicate defined over a set of constants each modeling one parameter. The predicate appears only non-negated in both all *preconditions* and all *effects*. Initial and goal states define the non-negated *have* predicate over the initially known and respectively required parameters. For example,

considering the web service in the example from section 2: *getVerpProp(textualWord)* that provides the *person*, *tense*, *number* and *mood*. The translated PDDL for this service is the following:

```
(:action getVerbProp
  :precondition
    (have textualWord)
  :effect
    (and (have person) (have tense)
         (have number) (have mood))
)
```

Planning complexity is PSPACE-complete in the general case[10] but the WSC reduction to planning is made to a subclass of planning that has non-negated predicates (no *delete* rules). This explains the existence of the polynomial algorithm proposed in section 5.

4. Composition builder test generator

In Zou et al[5] and other works the *ICEBE05* tests[11] are used for comparing WSC solutions. This benchmark was created for one of the first competitions on automatic WSC in 2005, and from our experiments it is not computationally hard or relevant for the solution length. Later, Oh et al[12] introduces *WSBen* tool that was also largely used with better results, both for running time and for the solution length. However, in order show the efficiency of the polynomial time algorithm proposed in section 3 over the planning methods, a new set of tests was created with a special generator that reveals a much higher running time variation of different solvers.

On both ICEBE05 and WSBen generated tests the solution found using either planning techniques or the polynomial algorithm is of at most 12 Web Services out of at most tens of thousands of web services within a repository. Since it is expected that the running time would be highly influenced by the solution length, this is not sufficient to differentiate between solutions, so the following generator is proposed.

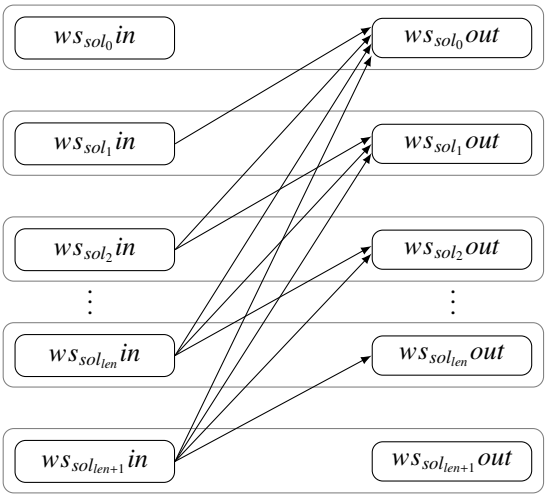
The generator creates a large repository of Web Services with random parameters and then chooses an ordered list of distinct Web Services that is later changed to contain a solution. It creates the user request as two fictive Web Services in the solution chain: the first is a service with output parameters as the user's request initially known parameters, and $\emptyset$ as input. Then for any service in the list in order it changes the **input** parameter list. The number of elements stays the same but the parameters are replaced by a (uniform) random choice of distinct parameters from union of the sets of **outputs** of previous services in the chain, including the user's request. The last extra 'service' will have no output parameters but the input is constructed similarly, and that input set is the goal required parameters.

Algorithm 1 is the summarized test generator algorithm. It uses four relevant input parameters that control the input and output size for the generated instance.

Algorithm 1 Solution-based test generator

/* **numWebServices** - number of Web Services, **parsPerService** - max parameter set size
numParameters - total distinct parameters, **numWSinSolution** - max solution length */
for $i \leftarrow 0, numWebServices$ **do** // create web service ws_i :
  $ws_i.in \leftarrow$ random parameter set // set size is random ∈ [1, parsPerService]
  $ws_i.out \leftarrow$ random parameter set // with values ∈ [1, numParameters]
$sol_0 \leftarrow 0$ // ws_0.out are initially known
for $i \leftarrow 1, numWSinSolution + 1$ **do**
  if $(i < numWSinSolution + 1)$ **then**
    $sol_i \leftarrow$ random service index
  else
    $sol_i \leftarrow 0$ // user request required
  $ws_{sol_i}.in \leftarrow$ random parameters from $\bigcup_{j=0}^{i-1} ws_{sol_j}.out$
// repository: $ws_1, ws_2, ..., ws_{numWebServices}$ and user (known, required) $\leftarrow (sol_0.out, sol_0.in)$

Figure 2: *Service inputs are constructed based on the output of previous services in the chain.*



5. Polynomial algorithm

The proposed algorithm makes valid calls to available services from the repository until the goal is reached: all the user required parameters are found. In order to implement this efficiently, a set of known parameters K is maintained that is first initialized with the user's initially known parameters. Let an *accessible* service be a service from the repository that has all input parameters included in the currently known parameters. The algorithm choses a new service that is *accessible* while it is possible. The output of the chosen service is added to K and the process continues. If at any point all goal parameters are completely included in the known parameter set, the search stops since a satisfying composition is found already. The search also stops if there are no *accessible* services and in this case the problem instance has no solution.

Algorithm 2 Chained matching algorithm

```
1:  function FindComposition(R, i, g)
2:      sol ← empty list // (of nodes)
3:      K ← i.O // set of known parameters
4:      ni ← newAccessibleService(R, sol, K)
5:      while ni ≠ NULL do
6:          sol.add(ni)
7:          K ← K ∪ ni.O
8:          if (g.I ⊆ K) then
9:              return sol // and exit
10:         ni ← newAccessibleService(R, sol, K)
11:     return NULL
12: function newAccessibleService(R, sol, K)
13:     for n ∈ R do
14:         if (n.I ⊆ K) ∧ (n ∉ sol) then return n
15:     return NULL
```

In Algorithm 2 we have the input R – the repository as a set of web services, *(i, g)* – the initially and goal parameters structured as services (with $i.I = \emptyset$ and $g.O = \emptyset$). *sol* is the ordered list of services in the constructed composition solution and K is the set of currently know parameters.

In order to implement *newAccessibleService()* efficiently some other structures are implemented:

- *inputParameterOf* – a map form any parameter to the set of all services that have this parameter as input.
- *unknownInputParameters* – a map form any service to the set of its input parameters that are currently unknown.
- *accessibleServices* – set of all services ws_i with $unknownInputParameters[ws_i] = \emptyset$.
- *userUnknownParameters* – set of user requested parameters that are unknown.

newAccessibleService() will return any element of *accessibleServices* set if it is not empty and *NULL* otherwise. If the set is empty, and the *userUnknownParameters* set is not empty, the problem instance has no solution. This is correct because we mark all parameters that are found on any invocation and once found a parameter is never forgotten. Moreover, if - for any reason - a service is invoked but it does not add any new parameter, that has no negative effect anywhere in the problem formulated as in section 2.

The data structures have to be updated after a service call. For any of the web service's output parameters, if that parameter was not in K (known) set, then it is a new learned parameter and this will only happen once. In this case we iterate through all the services in *inputParameterOf* that parameter and remove it from *unknownInputParameters* for the service. This does not increase the time complexity since any parameter is found only once and then added to K. When removing from *unknownInputParameters* of a web service, we check if there are still any unknown parameters for that service. If there are none, we add the service to *accessibleServices*.

Also, when a service is called it is permanently removed from *accessibleServices*: it will never add any benefit to call it again. *userUnknownParameters* is also updated when a new parameter is learned to enable constant time complexity of termination condition.

If we use hash table containers, e.g., hash set and hash map we can obtain a complexity of **O(1)** for a *newAccessibleService()* call and the same when checking if user goal is reached. The algorithm time complexity in this case is **O(N+M)** where **N** is the number of web services and **M** is the sum of their number of parameters, that is *linear* relative to problem input data size. This is because any service is called at most once, the choice of what service to call is implemented in **O(1)**, and any parameter is learned only once, even if it is in the output of more then one called service. When learning one parameter, all services that have it as input are iterated and processed in **O(1)** and the total number of these iterations can be at most **M**, because:

$$\sum_{ws_i \in R} |ws_i.I| = \sum_{p \in P} |inputParameterOf[p]| \leq \mathbf{M}\text{ , where: } P = \bigcup_{ws_i \in R} ws_i.I \cup ws_i.O\text{, the set of all parameters.}$$

6. Heuristics for solution length

Let **WSC-MinWS** be the problem defined in section 2 with the additional constraint of providing the solution with the *minimum* number of Web Services out of all possible compositions.

If **WSC** is easily solvable in polynomial time, **WSC-MinWS** is **NP-Complete**. This can be proven by reducing well known **Set Cover** to **WSC-MinWS**. Such a reduction can be found in [13]. However **WSC-MinWS** is closer to practical applications where cost, reliability, throughput of services have to be consider to provide a favorable composition solution thus the interest in it. For this purpose we propose a simple, score-based heuristic for reducing the solution length that does not significantly increase the running time.

Each distinct parameter and Web Service is associated with a positive floating-point score that approximates the expected benefit of finding that parameter or calling that Web Service. First, the scores are calculated for all parameters and services. Then the composition algorithm follows as described with the only difference that when one service is chosen from the currently accessible services the one with the highest score is chosen. The set is simply replaced by a set ordered decreasingly by the score values, and the time complexity is increased from **O(1)** to **O(logN)** if using Binary Search Tree or Heap structures.

The score-assigning algorithm follows two key principles:

I. Service score cumulates the score of its output parameters. This is because the score is a measure of the benefit of choosing that one service to call next, not the possibility of calling it, so the output parameters are prioritary to input parameters. Since the output parameters can be used further independently **sum** of scores is suitable.

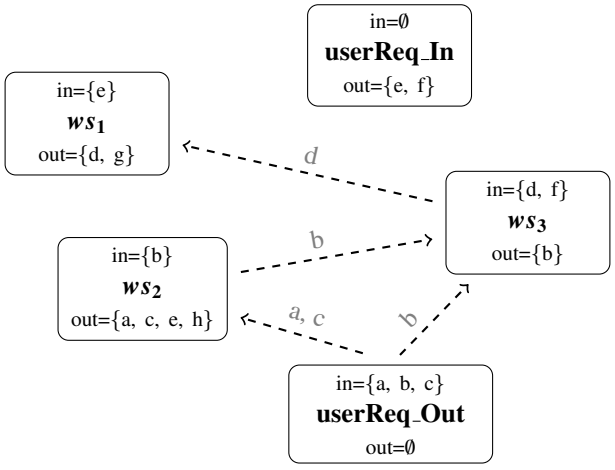
II. Parameter score cumulates the score of all the services that is input for. This is the other way around: if one parameter is found than any service that has it as input could possibly become accessible. This is a good estimate

although services might have other harder-to-reach parameters as input and remain unaccessible. Intuitively the score *"flows"* from parameters to services and from services to parameters. The initial score is associated to the user goal parameters that can be assigned with any positive constant number (e.g. each goal parameter ← 1)

Working with the above score implications can result in cycles since one parameter can play the role of both input and output. Some parameter score might increase the score of services that increase the score of parameters and so on and it is possible to reach the initial parameters on this path. It is impossible to compute all the scores in this manner so a limitation has to be imposed. Our chosen trade-off is to process any service only once. The algorithm first assigns score to parameters in the user request. Then for any service that provides any of these parameters, we process that service by calculating the score of that service. While doing this, some output parameters of that service might have no score assigned yet, and we handle them as having score 0. Next, the score of its input parameters is increased with service score divided by the number of input parameters. Some new services might output these parameters so they are added to the processing queue. Then the next service from the queue is processed and so on until the queue is empty.

At the end of the iteration of all services that could be reached, the sum of output parameters score is computed again for each service. This is a simple but effective improvement since at the time of first processing of each service some parameters might had lower score then at the end. This can be extended further in several ways, for example: if service score increases at the last step, we can increase parameters score accordingly. For simplicity and performance concerns only the first step was implemented.

Figure 3: *Score propagation from user required parameters towards accessible services*



For the example in figure 3 the first scores will be assigned to parameters *a, b, c* with the value of **1**. Services ws_2 and ws_3 are added to the queue since they produce at least one of these parameters. For service ws_2 the score is now **2** which is the sum of the scores of *a*, *c*, *e* and *h* that are **1**, **1**, **0** and **0**. Also *b*'s score is increased by **2** thus reaching **3**.

Service ws_3 is processed next and assigned the score of *b* that is **3**. *d* and *f* score is assigned to **3/2 = 1.5**, and ws_1 is added to the queue and processed next. Its score is the sum of *d* and *g* that is **1.5+0=1.5**. *e* score is **1.5/1 = 1.5** and the queue is empty.

At the end all services score are recalculated as the sum of their output parameters and in our case this changes only the score of ws_2 that is now including nonzero score of *e* and reaches **3.5**.

The Algorithm 2 follows the score allocation algorithm with the accessible services set sorted decreasingly by these final scores. However the provided solution can possibly be shortened further by a number of different tests. These have been proven important in practice.

One first observation is that any service that outputs no new parameter relative to previous services output in the chain is removed. Also, any output parameter that is not used by further services or the final user request is useless and thus services that output only this type of parameters can also be removed. To include all the corner cases here, we iterate all services in the solution chain, keeping two information for all parameters: if there is any service already processed that provides the parameter as output (if it is already provided), and how many unprocessed-yet services

require each parameter as their input (if it will be required further). In order implement this efficiently, the solution chain has to be iterated twice: the first time all parameter's input appearances are counted. On the second iteration this counter can be simply decreased on any new input appearance; the remaining counter will represent the appearances of a parameter in further services. This way, the time complexity of the algorithm is not increased, if proper data structures are used, like hash table containers. Finally, one service is useful if it outputs at least one parameter that is not provided by previous services but is required by further services in the chain, or in the final user required parameters. If one service is not useful by this definition, it can be dropped from the composition without invalidating it. These final steps have been found to be effective on benchmarks reducing the solution length on tests from two different benchmarks.

7. Results

The performance of our solution is compared with the planning based approach described in Zou et al[5] used with two different solvers: GraphPlan[6] and FastForward[7]. Each of the three solutions is executed on tests from three sources: *ICEBE'05* competition[11], WSBen[12] and tests from the generator in section 4. Two metrics are considered: the **running time** to deliver a solution and the **solution length**, that is the number of web services in the composition. All solutions provided are validated with a separate program that verifies the accessibility of each service in the composition in order and the provisioning of all user required parameters at the end. Results on each benchmark are presented on separate tables.

Table 1: *Results on a selection of ICEBE'05 tests*

$\|R\|$	file name	Algorithm 2		GraphPlan		Fast-Fwd	
		time (*s*)	length	time (*s*)	length	time (*s*)	length
2656	composition1-50-16	0.46	2	0.3	2	2.99	2
4156	composition1-100-4	0.26	4	0.64	4	2.6	4
5356	composition2-50-32	2.1	7	6.6	7	14.3	7
8356	composition2-100-32	3.6	7	3.0	7	22.1	7

$|R|$ is the number of web services in the *.WSDL* file or the repository size. Time is measured in seconds and *len* column specifies the length of the provided composition in number of web services.

The *ICEBE'05* tests are structured in two sets: *composition1* and *composition2* where the second is computationally harder. Our first two tests are from *composition1* and the last two are from *composition2*, where we can clearly see an increase in the running time of all solutions. Another important parameter of the tests is the number of parameters per service. The selected tests different choices for this parameter, specified in the last number of the file name (4, 16, 32). Also, each file in *ICEBE'05* provides *11* user requests from which one is selected randomly. The running times are compared on only this selected query that is the same for the three solutions.

The conclusion from *ICEBE'05* results is that *Fast-Forward* times are significantly higher especially on larger repositories and *GraphPlan* has about the same performance as our polynomial solution. On all tests, the solution length is the same but interestingly, the solutions are not identical and each of the three solutions provides almost totally disjoint compositions, with only a few services in common.

The *WSBen* tests are generated with the *Scale-Free* option. Other options (*Small World* or *Random*) and other parameter changes did not reveal significantly different behavior. As seen in Table 2, the running times are generally smaller relative to *ICEBE'05 composition-2* on the same repository size, thus the computational complexity is not higher. *Fast-Forward* is again the slowest and even fails on repositories with more than ten thousand services. To make *Fast-Forward* and even *GraphPlan* run on large repositories several limit specifying constants had to be increased in the code, like array limits and maximum number of operators. But even after this change *Fast-Forward* could not run on all tests in our experiments and failed with parsing errors. On *WSBen*, *GraphPlan* running times are closer to *Fast-Forward* than to our polynomial solution which is again the fastest.

Table 2: *Results on a selection of WSBen (Scale-Free option)*

$\|R\|$	Algorithm 2		GraphPlan		Fast-Fwd	
	time (s)	length	time (s)	length	time (s)	length
300	0.03	9	0.08	9	0.07	9
1000	0.1	7	0.3	11	1.4	6
5000	0.7	8	2.7	13	4.6	6
10000	1.7	9	3.6	15	error	?

But more important and different from previous results, *WSBen* tool revealed significant information about the provided solution length. *GraphPlan* is faster than *Fast-Forward* but it provides a much longer solution, our polynomial algorithm provides a solution almost as short as *Fast-Forward* but much faster. This was not the initial case as without the heuristics in section 6 and the final improvements added to reduce the solution length, the algorithm would reveal solutions that are roughly closer to the ones provided by *GraphPlan*.

Table 3: *Results on tests generated by algorithm 1*

$\|R\|$	solution length	parameters per service	Algorithm 2		GraphPlan		Fast-Fwd	
			time (s)	length	time (s)	length	time (s)	length
300	100	15	0.07	50	1.5	51	3.3	50
300	100	40	0.2	95	61	98	43	95
200	150	70	0.3	141	>3 hours	?	22 min	141
1000	200	50	1.6	130	39.9	133	error	?
1000	500	20	0.5	314	>3 hours	?	error	?
1000	100	20	0.4	86	4.8	87	error	?

The new generated tests revealed much more information as seen in Table 3, for comparing not only our solution but even the two planners relative behavior. First, more test parameters are to consider than just the repository size, and the most important is the *desired solution length* represented in the second column. This is the number of services chained to build a valid solution as in section 4, but it is not necessary that all this services are strictly required. Often shorter valid compositions are found because one service's output parameters might not be chosen in the random subset of parameters; that constitutes the input of next service. The easiest way to decrease the gap between the *desired solution length* and length of discovered solutions is to increase the number of parameters per service, shown in the third column. Each service's input and output parameters sets will have a size of a uniform random between 1 and this number of elements. Increasing this eventually increases the probability of one service being strictly dependent of a previous service.

For example, the first two tests have *300* repository size, and both have *desired solution length* set as *100* but the first has *15* maximum parameters per service while the second has *40*. This produces a significant difference in the resulting compositions: from a solution of length of only *50* for the first test the length is increased to *95* on the second. Last three tests are generated with different parameter values to provide long compositions both by increasing the *desired solution length* and the maximum *parameters per service*, and we can observe the solutions to be found from *86* to at most *314* web services. The observed solution length has the highest influence in the running times of planning based solutions. For more then 1-2 hundred services required in the composition, both planners simply block for hours or provide errors while the polynomial solution runs in at most a few seconds.

The test with repository size of *200* was very useful to test the solution shortening improvements: initially the heuristic provided a *148* services-long solution that only after the improvements was reduced first to *143* and finally *141*; that is interestingly exactly as short as the solution of *Fast-Forward*. Also, this test reveals a huge difference in the running times: from *0.3* seconds for the polynomial algorithm to *22* minutes for *Fast-Forward*. *GraphPlan* was stopped after *3* hours of running without providing any solution.

8. Conclusion

Our polynomial solution is simple yet much more efficient than the planning-based approaches, that are one of the most used to solve the Automatic Web Services Composition problem. It scales not only on the large repository size but even more important, when only long compositions exist. The length of the provided solution was at least as short as the planning based solutions with the only exception of a few tests from *WSBen* where at most *2* extra services were used. The proposed test generator can provide instances with more relevant results to compare between planners.

Future work can improve both the heuristic and solution shortening algorithm. At least, the scores can be updated dynamically on building the composition to reduce the importance of already found parameters in further decisions. Also we should consider pairs or combination of parameters together for scores since we expect this to provide more in-depth analysis of reachability for further services. The test generator algorithm can also be improved to require a smaller number of parameters per service yet still imply a high service dependency, since random choices of parameters is less efficient for this purpose then initially expected.

Practically much more important, the service description and composition language has to be extended to enable more general versions of the **WSC** problem. The proposed algorithm has to be adapted to support this. Such extensions can include semantic reasoning, concept similarity and more expressive input from the user. Quality of Service (QoS), services cost, reliability, throughput and more aspects can be taken into account, eventually by affecting the heuristic scores of parameters and services. Even if this is a very different work direction that leads the applicability to the next level, our hope and expectation is that the polynomial algorithm can be adapted to such a different **WSC** problem definition.

Acknowledgements

This work is partly funded from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 692178. This work was also partly supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI - UEFISCDI, project number 10/2016, within PNCDI III.

References

1. J. Rao and X. Su, "A survey of automated web service composition methods," in *International Workshop on Semantic Web Services and Web Process Composition*. Springer, 2004, pp. 43–54.
2. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.
3. J. Nitzsche, T. Van Lessen, D. Karastoyanova, and F. Leymann, "Bpel for semantic web services (bpel4sws)," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2007, pp. 179–188.
4. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne *et al.*, "Owl-s: Semantic markup for web services," *W3C member submission*, vol. 22, pp. 2007–04, 2004.
5. G. Zou, Y. Gan, Y. Chen, and B. Zhang, "Dynamic composition of web services using efficient planners in large-scale service repository," *Knowledge-Based Systems*, vol. 62, pp. 98–112, 2014.
6. A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.
7. J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, p. 57, 2001.
8. R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.
9. D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language," 1998.
10. T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1-2, pp. 165–204, 1994.
11. M. B. Blake, K. C. Tsui, and A. Wombacher, "The eee-05 challenge: A new web service discovery and composition competition," in *e-Technology, e-Commerce and e-Service, 2005. EEE'05. Proceedings. The 2005 IEEE International Conference on*. IEEE, 2005.
12. S.-C. Oh, H. Kil, D. Lee, and S. R. Kumara, "Wsben: A web services discovery and composition benchmark," in *Web Services, 2006. ICWS'06. International Conference on*. IEEE, 2006, pp. 239–248.
13. S. C. Geyik, B. K. Szymanski, and P. Zerfos, "Robust dynamic service composition in sensor networks," *IEEE Transactions on Services Computing*, vol. 6, no. 4, pp. 560–572, 2013.