Appendix A Additional Examples

A.1 Task Specifications

Table 1 is an extension of the specifications table included in Section 6, to include all the tasks that are part of the motivating scenario. To achieve a more condensed language-independent representation, the following simplifications are adopted:

– Only the HasInput variables and precondition predicates within the action precondition axioms are included.
– Only the HasOutput variables and postcondition predicates within the state update axioms are included.
– Only the accident cause is included for each qualification.
– Causal relationships are not included, instead ramifications are represented through the effect predicate they incur.

Files containing WSSL/XML specifications for all tasks are available at http://www.csd.uoc.gr/∼gmparg/wssl-sdf.

A.2 Example Heuristic Encoding

A possible heuristic encoding that can be used to created a suitable SBA for the motivating scenario is shown in Listing 1 in FLUX Prolog. Using this encoding, we can yield a plan that follows the process shown in Section 1 by executing the FLUX query `motiv_plan([hasinput(username, gpsactive(username)], P, Z_PR)`. Note that *dummy* denotes the execution of no service, in case change(prefs) is false. We also assume there is at most one execution of Set Preferences.

Table 1 Task specifications for the motivating scenario

Service	**Inputs**	**Outputs**	**Qualifications**
Get Location	username	user_location	GeolocError
Get Restaurants	user_location	rest_list	-
Get Preferences	username	pref_list	-
Set Preferences	new_prefs	pref_list	-
Filter List	rest_list, pref_list	filtered_list	-
Choose and Book	filtered_list	booking	BookingError
Get Best Route	booking, user_location	route	RoutingError
Service	**Preconditions**	**Postconditions**	**Ramifications**
Get Location	GeolocActive(username)	Located(username)	-
Get Restaurants	Located(username)	Verified(rest_list)	-
Get Preferences	-	Verified(pref_list)	-
Set Preferences	UpdRequest(username)	Updated(pref_list)	¬UpdRequest(username)
Filter List	Verified(rest_list), Verified(pref_list)	Verified(filtered_list)	-
Choose and Book	Verified(filtered_list), ¬UpdRequest(username)	Verified(booking)	RouteRequest(booking)
Get Best Route	RouteRequest(booking)	Verified(route)	¬RouteRequest(booking)

Listing 1 Heuristic encoding for the motivating scenario

```
motiv_plan(Z,[A|P],Z_PR) :- A1=getlocation, A2=getrestlist, A3=getprefs, A=and(A1;A2, A3),
          poss_and(A1;A2,A3,Z), state_update_and(Z,A1;A2,A3,Z_1), assist_plan1(Z_1,P,Z_PR).

motiv_plan1(Z,[A|P],Z_PR) :- A=filterlist, poss(A,Z), state_update(Z,A,Z_1)
                                                        , assist_plan2(Z_1,P,Z_PR).
motiv_plan2(Z,A,Z_PR) :- F=change(prefs), A1=setprefs, A2=filterlist, A3=dummy,
     A=if(F,A1;A2,A3), poss_if(F,A1;A2,A3,Z), state_update_if(Z,F,A1;A2,A3,Z_PR)
                                                            , assist_plan3(Z_1,P,Z_PR).

motiv_plan3(Z,[A|P],Z_PR) :- A=chooseandbook, poss(A,Z), state_update(Z,A,Z_1)
                                                            , assist_plan4(Z_1,P,Z_PR).
motiv_plan4(Z,[A|P],Z_PR) :- A=getbestroute, poss(A,Z), state_update(Z,A,Z_1).
```

Appendix B Additions to QoS-related Aspects

B.1 QoS Profile Correctness for Set Operators

In order to take into account attributes that may take multiple values, additional set operators are defined for constraints: $\{\in, \notin, \subset, \subseteq, \supset, \supseteq, \equiv\}$. Since the table included in Section 4.1 does not cover these operators, we need to explore correctness violation cases when constraints involve set operators. All possible violation cases for such constraints are presented in Table 2.

B.2 QoS Aggregation Algorithm

Algorithm 1 realises QoS aggregation for the case where no knowledge of execution path probabilities is available and relies on the aggregation formulas in Section 4.1.1. It is essentially a traversal of a tree-like workflow, in order to find all attribute values related to global QoS goals; hence, its complexity is $\mathcal{O}(n)$, where n is the number of tasks.

ALGORITHM 1: Aggregating QoS values in BPMN processes

```
input  : A BPMN2 process (in XML), WSSL/XML
         documents for all participating services and the
         QoS attribute to be aggregated
output: The aggregated QoS value
if attribute is pattern-independent then
    nodeList ← GetNodes();
    foreach node n in nodeList do
        if node is a task then
            GetQoSValue();
        end
    end
    apply aggregation function on collected QoS values;
end
else
    aggValue ← AggregatePath(n);
end

AggregatePath(n)
firstNode ← GetFirstNode(n);
if firstNode is SplitNode then
    outC ← GetOutgoing();
    foreach node c in outC do
        aggValue ← AggregatePath(c);
    end
end
else
    GetQoSValue();
end
apply aggregation function on collected QoS values;
```

Table 2 Violation of QoS profile correctness (set operators)

Constraint 1	Constraint 2	Sets Relation
$x \in set_1$	$x \in set_2$	$set_1 \cap set_2 = \emptyset$
$x \in set_1$	$x \subset set_2$	$set_1 \equiv set_2$
$x \in set_1$	$x \subset set_2$ or $x \subseteq set_2$	$set_1 \cap set_2 = \emptyset$
$x \in set_1$	$x \supset set_2$ or $x \supseteq set_2$	$set_2 \neq \emptyset$
$x \notin set_1$	$x \subset set_2$ or $x \subseteq set_2$	$set_1 \equiv set_2$
$x \notin set_1$	$x \supset set_2$ or $x \supseteq set_2$	$set_2 \neq \emptyset$
$x \subset set_1$	$x \subset set_2$ or $x \subseteq set_2$	$set_1 \cap set_2 = \emptyset$
$x \subset set_1$	$x \supset set_2$ or $x \supseteq set_2$	$set_1 \cap set_2 = \emptyset$ or $set_2 \supseteq set_1$
$x \subseteq set_1$	$x \subseteq set_2$	$set_1 \cap set_2 = \emptyset$
$x \subseteq set_1$	$x \supset set_2$	$set_1 \cap set_2 = \emptyset$ or $set_2 \supseteq set_1$
$x \subseteq set_1$	$x \supseteq set_2$	$set_1 \cap set_2 = \emptyset$ or $set_2 \supset set_1$
$x \supset set_1$	$x \supseteq set_2$	$set_1 \equiv set_2$

Appendix C Additions to Evaluation

C.1 Comparison Process

To be able to compare among execution times calculated using different CPUs, we decided to normalise values using performance benchmarks available in [5]. These benchmarks assign an *average mark* to each CPU which is an indicator of the achieved performance and allows comparison: a CPU achieving a 2X mark is twice as fast as a CPU achieving an X mark. Admittedly, CPUs are not the only system characteristic that affects performance; anything from memory to operating systems to compiler setups can affect execution time. However, a CPU-based comparison can, at the very least, provide a crude estimate of what execution times would be achieved if all experiments were run on the same CPU.

Our experiments were performed on an Intel® Core™ i7-740QM processor running at 1.73GHz. This CPU has an average mark of 3207. [6] used an Intel® Xeon® Dual CPU X5450 running at 3.0GHz, which has an average mark of 7592. To that end, execution times reported in [6] are normalised using a correction factor of 7592/3207=2.36. For the QoS-based selection experiments in [2], a setup of two AMD Athlon™64 FX-74 running at 1.80GHz was used, which has an average mark of 2947. The correction factor for this case is 2947/3207=0.92. [2] relies on the COCOA composition system [3], which was evaluated using an Intel® Pentium™4 running at 2.80GHz. This CPU has an average mark of 315, hence the correction factor is 315/3207 = 0.098. Finally, [2] uses the EASY discovery system [4], which was evaluated using an Intel® Core™Duo T2050 processor running at 1.60GHz. This CPU has an average mark of 704, hence the correction factor is 704/3207 = 0.22.

As noted in the main paper, the execution times achieved by [6] are significantly higher, even if we do not use normalisation and disregard the fact that the evaluation was run on a really powerful system. On the other hand, the experiments in [3] and [4] were run on considerably slower machines, but due to the performed normalisation reported values are adapted to be 10 and 5 times faster, respectively, than what was originally calculated.

C.2 Performance of Composition Planning

In order to evaluate functional composition via planning with WSSL, we ran a series of experiments, calculating the time needed for the planner to produce a valid service composition plan, given a set of services, an initial state and a goal state. The parameters that are of interest in this evaluation are the following:

1. *Specification repository size:* we assume that different implementations of the same functionality are represented by a single specification, for the purposes of functional planning. In the experiments, we vary the size from 5 to 500 distinct specifications (i.e. functionalities), while each specification consists of 1-2 IOPE quads, representing an average specification length. Depending on how many actual implementations one can assume each specification corresponds to, the specification repository may map to a service repository containing thousands of concrete services.
2. *Inclusion of ramifications:* adding a ramification increases specification size due to the inclusion of a causal relationship. When ramifications are taken into account in experiments, a ramification is added to each specification, linking a primary effect to a newly added secondary via a causal relationship. Half of these ramifications are required to achieve the plan goal, representing a case of higher than average complexity.
3. *Complexity of composition heuristics:* this is defined in terms of the combination of control constructs that is taken into account.

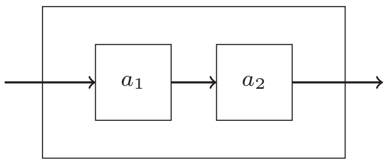

Fig. 1 Building block for sequential chains

We assume three cases of increasing complexity. The first case, shown in Fig. 1, contains sequential chains of services with one IOPE quad each, based on the getLocationofAddress and getDistanceBetweenLocations services in OWL-S TC [1]. For each pair of services, the output of the first one is the input of the second, while the postcondition of the first is required as a precondition for the second. All inputs, outputs, preconditions and postconditions are uniquely named. For this experiment, we increased chain length from 2 to 500; chain length variation is achieved by modifying the composition goal to include postconditions that require larger and larger chains to be achieved.

The second case contains sequential chains of pairs of services composed in parallel using the AND-Split/Join control construct, as shown in Fig. 2. Both services are customisations of getLocationofAddress in order to include two IOPE quads and follow the same conventions that concern linking and naming as in the previous case.

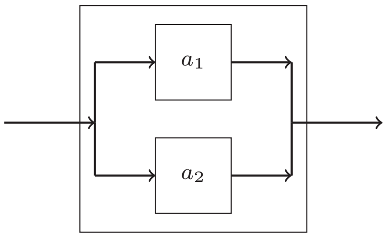

Fig. 2 Building block for chains of parallel executions

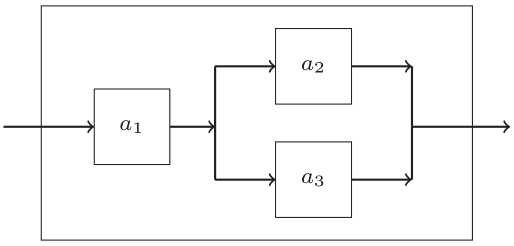

Fig. 3 Alternating between atomic services and parallel pairs

For this experiment, we ranged from a sequence of 1 parallel pair to a sequence of 250 parallel pairs.

The third case combines the first two, containing sequential chains of atomic services alternating with AND-Split/AND-Join pairs, as shown in Fig. 3. In this case, some services contain one IOPE quad, while others contain two, in order to achieve the described alternation. Again, the same linking and naming conventions apply. For this experiment, we varied sequence length from 2 to 332.

For all three cases, we calculated separately the cost of producing a single solution to the planning problem and all possible ones; the former is interesting in order to determine the least time required to generate a solution, while the latter corresponds to what is required for the first phase of WSSL/SDF. Note that for any given pair of services that can be executed in a state (i.e. their poss clauses succeed), any of the control constructs can be considered, since all foundational axioms evaluate to true; however, only the first one defined in the heuristic will be chosen, due to the declarative nature of FLUX. In our experiments, we chose to order heuristics so that sequence and AND-Split/Join are defined first; making any other choice (e.g., choose OR-Split/Join instead) would not affect evaluation results, since the cost of checking any foundational axiom is the same. The use of specific control constructs is expected to be dictated explicitly in heuristic encodings.

The results of the three experiments are shown in Figs. 4, 5 and 6. As evidenced in Fig. 4, searching for all possible planning problem solutions, instead of a single one, results in a 40% increase in computation time in the sequential case, which is reasonable and expected, given the cost of backtracking performed to find all solutions. The effect is more severe in the parallel case (Fig. 5), where we observe a twofold increase, due to the increase of backtracking points caused by the special poss_and and state_update_and rules that are employed by AND-Split/Join. Finally, as shown in Fig. 6, the most complex case results in a much less significant increase in computation time, 20% on average. This is explained by taking into account that the composition plan is even more complex, hence requiring more time to find a single solution; however the increase of backtracking points is not so severe, since we just add sequential steps to the parallel executions of the second experiment.

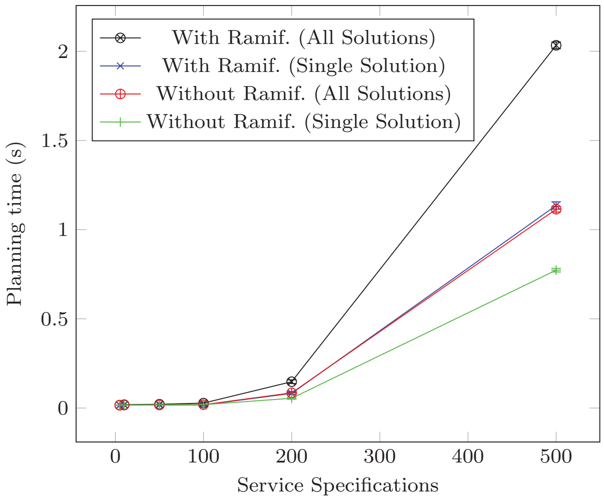

Fig. 4 Scalability results for sequential-only plans

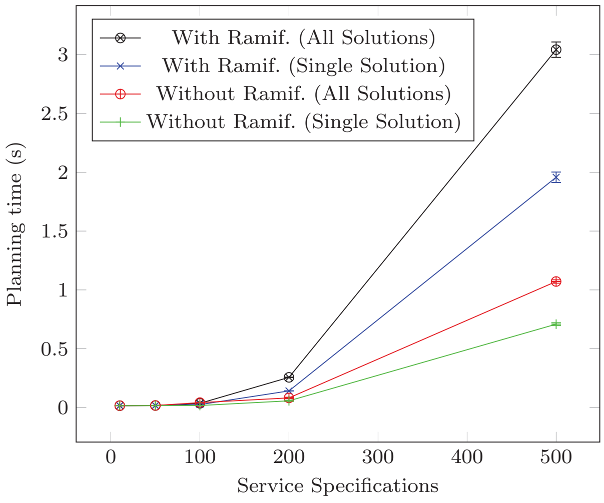

Fig. 5 Scalability results for parallel-only plans

As far as the effect of adding ramifications is concerned, the severity is inversely proportional to composition complexity. In the simplest case, there is an 85% increase in the time required to find all solutions; in the parallel-only case, the increase falls to 60%, while

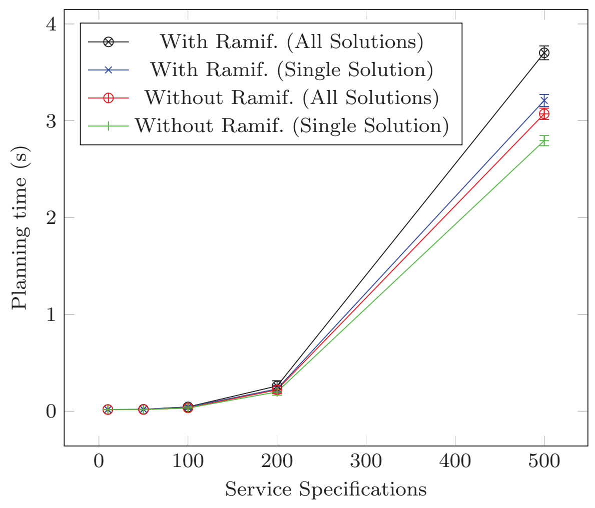

Fig. 6 Scalability results for sequential/parallel combination plans

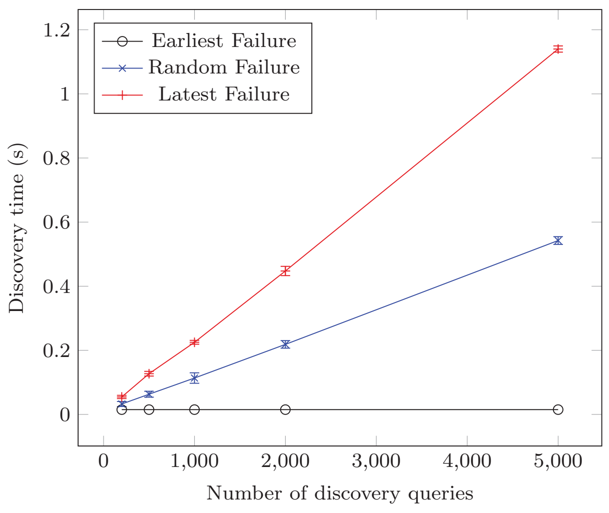

Fig. 7 Scalability results for functional discovery

in the most complex case, there is only a 30% increase. This is due to the fact that, in the simpler cases, the effect of the effort to find a matching causal relationship is more pronounced, since the amount of backtracking performed by the planner is low; in more complex cases, the complexity is already increased, rendering the addition of ramifications less significant.

In all cases, the increase is exponential with regard to the repository size, which is to be expected since the number of choice points grows exponentially. However, computation time is kept at reasonable levels (under 1.5 sec) for 300 specifications, even for the more complex cases. This breaking point is well above what is expected in real-world service design scenarios, keeping in mind that we are dealing with specification repositories and not ones containing WSDL descriptions of concrete services. Given also that a WSDL document is much smaller than a WSSL specification document, due to the limited amount of information described by WSDL (mainly interface details), a WSSL repository containing the same number of documents as a WSDL repository is actually much larger. The relative standard deviation for the planning experiment is around 2% and is due to the fact that the planner follows the same traversal path in each execution, when given the same planning problem.

Concerning knowledge states, the fact that FLUX employs negation as failure to prove knowledge (or lack thereof) of a fluent means that the cost of evaluating a knows or knows_not clause is equivalent to that of a holds or not_holds one. Hence, there is no need to conduct experiments that specifically employ knowledge clauses. On the other hand, incomplete state specification using even a few constraints can quickly result in non-terminating executions when attempting to find all solutions without employing restrictive enough heuristics: the planner can always infer that the goal is achieved under any possible action combination, provided that no constraint is violated. Hence, in such cases, heuristic encodings do not only assist in restricting plan space, but also render planning based on incomplete states realisable.

C.3 Performance of Functional Discovery

In this experiment, specification size was fixed to 5 IOPE quads, a value that exceeds typical specification complexity: for instance, the most complex service in OWL-S TC has 8 inputs, 2 outputs, but only a single precondition and a single postcondition. Note that, instead of repository size, we vary the equivalent parameter of discovery queries, from 200 to 5000 independent queries. Note that the computation cost of 5000 such queries is equivalent to checking 10 distinct task specifications against a repository of 500 service specifications. We examine three different cases, as follows:

1. Discovery fails at the earliest point for each specification, due to not finding a match of the first precondition fluent.
2. Discovery fails at the latest point, due to not finding a match of the last postcondition fluent.
3. Discovery fails at a random point between the earliest and latest points, inclusive.

The results, presented in Fig. 7, show that execution time increases linearly with the number of queries, because process complexity is analogous to the number of fluents (representing IOPEs) that need to be compared. Note that in the case of earliest failure, computation time is less than the minimum recordable time in ECLiPSe (15 msec); however, we can expect that the values, albeit too small to be recorded, still increase linearly. At the worst (and improbable) case of discovery failing at the latest point for all 5000 queries, execution time peaks at roughly 1 sec; at the average random case, 5000 discovery queries require about 0.5 sec, or 0.1 msec per query on average, proving that it is scalable enough to be considered for complex composition scenarios (see also Section C.6). Relative standard deviation is insignificant, around 2-3% in the latest failure case, while increasing slightly in the random case, to values of 5-10% due to the fact that the included service specifications exhibit higher variability.

C.4 Performance of Pruning

To investigate performance of the pruning process, we consider an exceptionally complex setting of 1000 different sequential plans with 100 tasks per plan and 20 local QoS goals, while we vary the number of available implementations for each task from 5 to 100 and pruning success rate (i.e. what percent of plans are pruned) from 10% to 100%, to cover all possible pruning cases. Note that some of these values exceed what is generally expected, so that we obtain a clearer view of pruning behaviour, even in extreme situations. The results in Fig. 8 show computation times that are one order of magnitude lower than the planning process, with a linear increase in execution time. Higher pruning success rates decrease execution time, since entire plans may be discarded early, which explains also the low execution time for the case of 100% success rate: pruning is always successful, leading to quickly discarding all of the plans. Relative standard deviation ranges from 9% to 14% for this experiment, increasing with the number of implementations and as success rate decreases. These values are attributed to the fact that, in each execution, the cause of pruning, i.e. the task for which all available implementations violate a local QoS goal, is randomly chosen; having more implementations to choose from leads to a larger variation, either because plans have not been pruned yet, or if the total number of implementations is higher.

C.5 Performance of Ranking

Since the extended plan ranking process is not computationally complex, instead of evaluating execution time scalability, we investigate optimality in terms of finding heuristics that lead to extended plans that not only satisfy all goals but do so in the best possible way, in terms

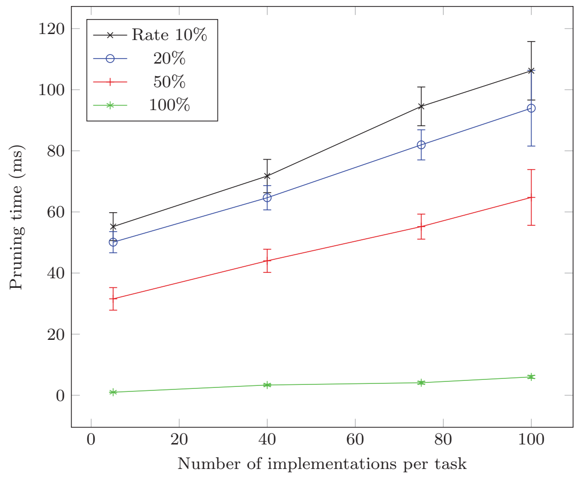

Fig. 8 Scalability results for pruning

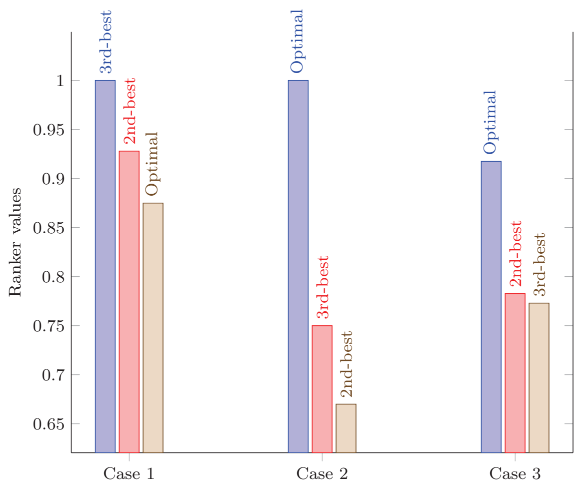

Fig. 9 Optimality results for ranking

of the criteria that follow. We use the motivating scenario as a basis and consider the process in Fig. 1 as the optimal plan, a result of manually designing a service composition that satisfies all functional goals. We examine the following three cases: only workflow-based ranking criteria, only problem-dependent ranking heuristics and combination of both. The workflow-based criteria we employ are maximum execution path length and total number of tasks; the problem-dependent heuristics are preferring plans that support the choice of setting preferences and additionally preferring plans that support both parallelisation of get location/restaurant list and get preferences; these enable the composite process to cover all possible cases that are mentioned in the motivating scenario.

Fig. 9 shows the top-three performance for each case; in all cases, the actual three most optimal extended plans are ranked at the top (the one in Fig. 1, plus variations that do not achieve a single criterion), but not always with the correct order. In Case 1 (only workflow-based criteria), the order is reversed, with the optimal plan ranked third, deviating 12.5% from the highest ranking value. In Case 2 (only problem-dependent heuristics), the optimal plan is ranked correctly at the top; however, the other two results are flipped. Perfect optimality is achieved in Case 3 where both types of criteria are taken into account. These results show that, for the case of the motivating scenario, workflow-based ranking is not enough to yield perfect or near-perfect optimality, whereas problem-specific ranking heuristics can correctly yield the top-ranked plan, even without taking problem-independent criteria into account.

C.6 Overall performance

In this section, we present an experiment that targets overall performance of the proposed framework. While previous experiments focused on investigating performance of each framework phase by exploring wide ranges of parameters of interest to test the limitations of WSSL/SDF, this experiment considers the average case (compared to the previous ones) of a repository containing 600 services representing 200 distinct specifications (3 implementations for each task). Services in the repository are variations of the SendPayment service in the WS-Dream dataset[1] (which also provides actual QoS measurements from deployment). These variations realise 1 to 5 payment transactions, so that we result in services with 1 to 5 IOPEs. Specifically, 90 specifications (45% of the repository) realise 90 distinct payment transactions, labeled from 1 to 90; 60 specifications (30% of the repository) realise pairs of transactions (those numbered 1 and 2, those numbered 2 and 3, and so on); 30 specifications (15% of the repository) realise three transactions each; 15 specifications (7% of the repository) realise four transactions each; and 5 specifications (3% of the repository) realise five transactions each. For each of the 5 cases, we choose an increasing number of payment transactions that need to be processed, in order to result in composition problems of varying complexity: Case 1 involves 10 consecutive transactions numbered 40 to 49, Case 2 involves 11 (till 50), Case 3 involves 12 (till 51), Case 4 involves 13 (till 52) and Case 5 involves 14 (till 53). these values represent a realistic distribution of quad numbers. Additionally, a causal relationship is added to half of the specifications.

For each of these cases, we calculate the total runtime for all four phases of WSSL/SDF. Beginning with the planning phase, the planner results in the following sets:

- Case 1: 3 plans of length 9
- Case 2: 5 plans of length 9-10
- Case 3: 20 plans of length 9-12
- Case 4: 112 plans of length 10-14
- Case 5: 2167 plans of length 10-17

The observed variation in the number of plans is a direct consequence of the defined goals: from case to case the selected goal leads to longer chains, with more possible combinations for each pair in the chain. Following planning, we evaluate functional discovery for each case. We assume the median situation where the failure point is halfway through the specification. Given that each composition is a chain of services that is a subset of the repository and examining the different specifications that are involved in each plan, we calculate the maximum number of distinct specifications (and discovery runs) as follows. For Case 1, there are 10 distinct specifications that realise single transactions numbered 40 to 49; there are 10 specifications that realise these transactions, but in consecutive pairs (39 and 40, 40 and 41, and so on); similarly, there are 9 specifications that realise these transactions in groups of three, 8 in groups of four and 7 in groups of five. This results in 44 distinct specifications. However, Case 1 consists of only 3 abstract plans of length 9, meaning that there cannot be more than 27 distinct specifications taking part in these plans. For Case 2, there are 11 distinct specifications that realise single transactions numbered 40 to 50; there are 11 specifications that realise these transactions, but in consecutive pairs (39 and 40, 40 and 41, and so on); similarly, there are 10 specifications that realise these transactions in groups of three, 9 in groups of four and 8 in groups of five. This results in 49 distinct specifications at most. Following the same pattern for the next three cases, we result in 54 maximum distinct specifications for Case 3, 59 for Case 4 and 64 for Case 5. Considering a success rate that leads to 10% of abstract plans being discarded because no matching implementation could be discovered for all tasks of the plan, the following sets of extended plans are produced:

- Case 1: 3 plans with 9 tasks
- Case 2: 5 plans with 10 tasks
- Case 3: 18 plans with 11 tasks on average
- Case 4: 101 plans with 12 tasks on average
- Case 5: 1951 plans with 14 tasks on average

Conducting pruning based on 5 local QoS goals and ranking based on plan length and number of tasks per plan, we result in the following optimal extended plans that are fed to the final phase, the QoS-based selection process:

[1] http://wsdream.github.io

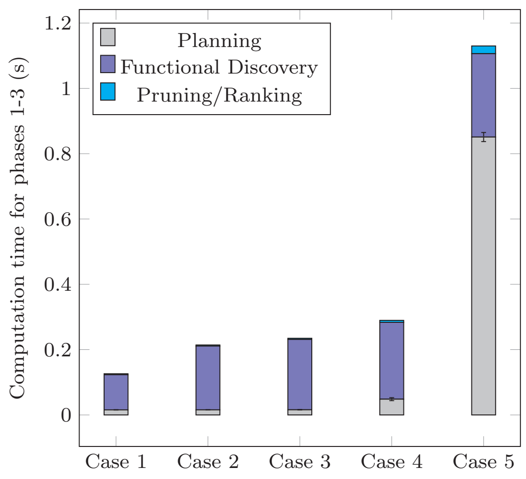

Fig. 10 Overall performance for the first three phases

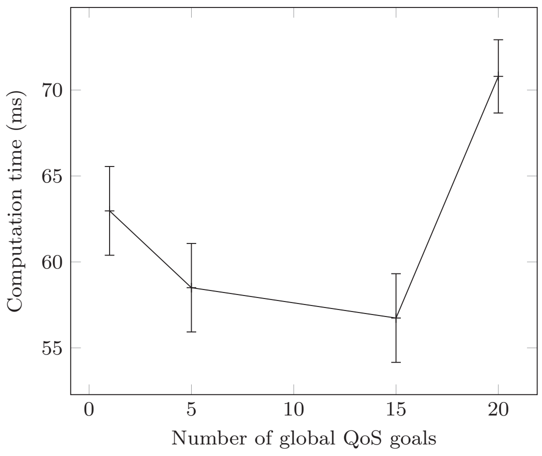

Fig. 11 Overall performance for the fourth phase (QoS-based selection)

– Cases 1-3: Sequential plan with 9 tasks, 3 implementations for each task and 3 QoS profiles for each implementation that survive pruning.
– Cases 4-5: Similar to the other cases, but with a sequence containing 10 tasks.

Fig. 10 shows the computation time for the first three phases, while Fig. 10 shows the computation time of QoS-based selection for the optimal extended plan, varying global QoS goals from 1 to 20. Fig. 12 combines the results of all four phases in a single graph. The results indicate that the majority of computation time, except in Case 5, is attributed to functional discovery, because we assume the worst case (maximum number of discovery runs per plan), as well as to QoS selection, which requires a similar amount of computation time on average. The pruning/ranking cost is negligible in comparison, since this phase has the least computational complexity. For QoS-based selection, there is no significant variation in computation time when we increase

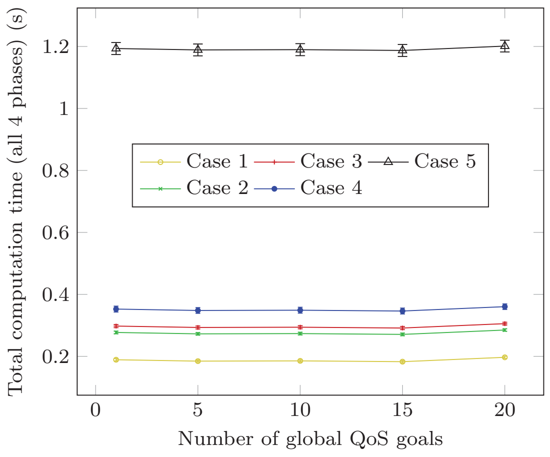

Fig. 12 Overall performance for all phases of WSSL/SDF

the number of global QoS goals. This behaviour implies a threshold after which the complexity of the MIP problem becomes more significant than the possibility of discarding candidate QoS offerings sooner, due to the increase in the number of constraints. Relative standard deviation is insignificant, averaging around 4%. In Case 5, a 20-time increase in the number of plans results in a similar increase in computation time, which peaks at 1.35 seconds, a rather satisfactory value for design time.

References

1. Klusch M, Khalid MA, Kapahnke P, Fries B, Vasileski M (2010) OWL-S TC: Service Retrieval Test Collection. Online (http://projects.semwebcentral.org/projects/owls-tc/)
2. Mabrouk NB, Beauche S, Kuznetsova E, Georgantas N, Issarny V (2009) QoS-Aware Service Composition in Dynamic Service Oriented Environments. In: Bacon J, Cooper BF (eds) Middleware 2009: Proceedings of the 10th International Middleware Conference, LNCS, vol 5896, Springer, pp 123–142
3. Mokhtar SB, Georgantas N, Issarny V (2007) COCOA: COnversation-based service COmposition in pervAsive computing environments with QoS support. Journal of Systems and Software 80(12):1941–1955
4. Mokhtar SB, Preuveneers D, Georgantas N, Issarny V, Berbers Y (2008) Easy: Efficient semantic service discovery in pervasive computing environments with qos and context support. Journal of Systems and Software 81(5):785–808, DOI DOI: 10.1016/j.jss.2007.07.030, URL http://www.sciencedirect.com/science/article/B6V0N-4PCXG9S-1/2/5a77d8d2987c35158348af8c5beacd99, software Process and Product Measurement
5. PassMark Software (2017) CPU Benchmarks. URL https://www.cpubenchmark.net/
6. Rosenberg F, Celikovic P, Michlmayr A, Leitner P, Dustdar S (2009) An End-to-End Approach for QoS-Aware Service Composition. In: Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference, IEEE Computer Society, Washington, DC, USA, EDOC '09, pp 151–160