Efficient Keyword Search for Building Service-Based Systems Based on Dynamic Programming

Qiang He[1,2(✉)], Rui Zhou[2], Xuyun Zhang[3], Yanchun Wang[2], Dayong Ye[2], Feifei Chen[4], Shiping Chen[5], John Grundy[6], and Yun Yang[2]

[1] State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China
[2] Swinburne University of Technology, Hawthorn, Australia
{qhe, rzhou, yanchunwang, dye, yyang}@swin.edu.au
[3] University of Auckland, Auckland, New Zealand
xuyun.zhang@auckland.ac.nz
[4] Federation University, Melbourne, Australia
feifei.chen@federation.edu.au
[5] Data61, CSIRO, Canberra, Australia
shiping.chen@data61.csiro.au
[6] Deakin University, Burwood, Australia
j.grundy@deakin.edu.au

Abstract. The advances in service-oriented architecture (SOA) have fueled the demand for building service-based systems (SBSs) by composing existing services. Finding appropriate component services is a key step during the process for building SBSs. However, existing approaches require that system engineers have detailed knowledge of SOA techniques, which is often too demanding. A recent approach has been proposed to address this issue. However, it suffers from poor efficiency, which is increasingly critical as the service repository continues to grow. To address this issue, this paper proposes KS3+, a new, highly efficient approach that allows a system engineer to query for a system solution with a few keywords that represent the required system tasks. Modeling the problem of answering such a keyword query as a dynamic programming problem, KS3+ can quickly find a system solution composed of services that perform the required system tasks. It offers an efficient paradigm that significantly reduces the time and effort during the process for building SBSs. The results of extensive experiments on a real-world web service dataset demonstrate the high efficiency and effectiveness of KS3+.

Keywords: Service oriented architecture · Service-based systems · Keyword search · Web services

M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 462–470, 2017.
https://doi.org/10.1007/978-3-319-69035-3_33

1 Introduction

The service-oriented architecture (SOA) has been widely employed by many enterprises to build service-based systems (SBSs) [1, 2]. The component services of an SBS collectively realize the functionality of the SBS, which are often offered as SaaS (Software-as-a-Service) to internal and external users in the cloud environment. The development and popularity of e-business, ecommerce, especially the pay-as-you-go business model promoted by cloud computing, have fueled the rapid growth of services and SBSs, shown by statistics published by programmableweb.com, a web service directory. The process for building an SBS consists of three phases: (1) System Planning: the system engineer empirically identifies and determines the system tasks, e.g., *flight ticket booking*, *hotel booking*, as well as the execution order of the tasks. (2) Service Discovery: the system engineer, through querying service repositories or service search engines, discovers multiple sets of composable services, each offering one of the required system tasks. (3) Service Selection: the system engineer selects one service from each set of candidate services to compose the target system that fulfills the multi-dimensional constraints and the optimization goal for the system quality, e.g., reliability, response time and cost.

The process above is complicated and requires detailed knowledge of sophisticated SOA techniques in different phases. It has become a major obstacle to broader applications of SOA. There has been a rapid increase in the need for an approach that assists system engineers in quickly finding *system solutions* for their SBSs, including which services to use and in what order they are composed, without going through the above complicated process [3].

We previously presented KS3 to tackle this challenge [4]. KS3 allows system engineers to query for system solutions by entering only a few keywords that represent the required system tasks. Such a *keyword query*, i.e., a query containing keywords that represent the required system tasks, is modeled as a constraint optimization problem and employs the integer programming technique to find system solutions. **However, KS3 suffers from extremely poor efficiency in processing queries on large web service repositories.** According to [4], it takes up to 100 s to answer queries on a repository with 20,000 web services. To address this issue, this paper proposes KS3+, a new, highly efficient approach for building SBSs also based on keyword search techniques.

2 Keyword Search Method

We discuss how KS3+ models keyword queries for system solutions and finds group Steiner trees [4] as answer trees to these keyword queries. We denote the set of keywords in a query Q as $\mathbf{K} = \{k_1, k_2, \ldots, k_l\}$ and use $\mathbf{k}$, $\mathbf{k_x}$, and $\mathbf{k_y}$ to denote a non-empty set of K where $\mathbf{k}, \mathbf{k_x}, \mathbf{k_y} \subseteq \mathbf{K}$. To represent a group Steiner tree that is rooted at node v and covers a set of keywords $\mathbf{k}$, we use $T(v, \mathbf{k})$. Thus, the group Steiner tree we look for in data graph $G(V, E)$ as answer to Q is $T(v, \mathbf{K})$ where $v \in V$ represents a web service and $e \in E$ represents the composability of two web services. For more details about G, see [4].

2.1 Dynamic Programming Model

In this research, a group Steiner tree $T(v, \mathbf{K})$ of height h (the length of the longest downward path from the root of the group Steiner tree to any leaf) can be found by expanding the group Steiner trees of heights $h = 0, 1, \ldots$, that cover $\mathbf{k} \subseteq \mathbf{k}$. Let $T(v, k)$ be a state in the dynamic programming model, and $w(T(v, \mathbf{k}))$ be the weight of $T(v, \mathbf{k})$, i.e., the total weight of the nodes in $T(v, k)$, the state-transition equation in the dynamic programming model is:

$$w(T(v,\mathbf{k})) = \min\left(w\left(T_g(v,\mathbf{k})\right), w(T_m(v,\mathbf{k}))\right) \tag{1}$$

$$w\left(T_g(v,\mathbf{k})\right) = \min_{u \in N(v)} \{w(T(u,\mathbf{k}) + u)\} \tag{2}$$

$$w(T_m(v,\mathbf{k})) = \min_{\substack{\mathbf{k}=\mathbf{k_1}\cup\mathbf{k_2} \\ \wedge \mathbf{k_1}\cap\mathbf{k_2}=\emptyset}} \{w(T(v,\mathbf{k_1}) + T(v,\mathbf{k_2}))\} \tag{3}$$

where "+" is an operation to merge a node into a tree or to merge two trees to a new tree, $N(v)$ is the set of node v's neighbors in G, i.e., $v \in G(V, E)$ and $e(u, v) \in E$. Equation (1) indicates that the weight of the a group Steiner tree $T(v, \mathbf{k})$ can be obtained by either of two cases, namely *tree growth*, i.e. Eq. (2), and *tree merging*, i.e. Eq. (3). As indicated by Eq. (2), the tree growth case is that $T_g(v, \mathbf{k})$ can be obtained by growing a node u from the minimum-weight subtree of $T(v, \mathbf{k})$ that is rooted at u (one of v's neighbors) and covers all keywords in $\mathbf{k}$. Equation (3) shows that, in the tree merging case, $T_m(v, \mathbf{k})$ can be obtained by merging two minimum-weight subtrees, both rooted at v, one covering $\mathbf{k_1}$ and the other covering $\mathbf{k_2}$ such that $\mathbf{k} = \mathbf{k_1} \cup \mathbf{k_2}$ and $\mathbf{k_1} \cap \mathbf{k_2} = \emptyset$.

2.2 Answering Keyword Queries

A keyword query Q contains a set of keywords, $\mathbf{K} = \{k_1, \ldots, k_l\}$. Based on Eqs. (1)–(3), KS3+ employs Algorithm 1 to find the minimum group Steiner tree as the answer to query Q_n. In line 1, Algorithm 1 initializes a priority queue of trees Q_T to be empty. The trees in Q_T are always sorted in ascending order by the total number of nodes in the trees, denoted by $|T|$. In lines 2–6, the algorithm locates nodes that contain individual keywords in $\mathbf{K}$. For each node v in G, $v \in V$, if v contains any keywords $\mathbf{k}$ in $\mathbf{K}$, $\mathbf{k} \subseteq \mathbf{k}$, the algorithm enqueues tree T(v, k) into Q_T. At this stage, for each such tree in Q_T, there is $|T(v, k)| = 1$ because there is only one node in each of the trees in Q_T. In lines 7–33, the algorithm iterates to dequeue trees from and enqueue trees into Q_T, and in the meantime grow them with Eq. (2) (lines 12–21) or merge them with Eq. (3) (lines 23–32) to find the minimum group Steiner tree $T(v, \mathbf{k})$, where $v \in V$ and $\mathbf{k} = \mathbf{K}$ (lines 9–11). Equation (2) is implemented by lines 12–21. Given a tree $T(v, \mathbf{k})$ just dequeued from Q_T (line 8), the algorithm considers all v's neighbors, denoted by u, and checks whether there is a tree $T(u, \mathbf{k})$ in Q_T that can be replaced with $T(v, \mathbf{k}) + u$, which contains the same set of keywords $\mathbf{k}$ but with fewer nodes (lines 12–17). If such a $T(u, \mathbf{k})$ does not exist in Q_T, $T(v, \mathbf{k}) + u$ is enqueued into Q_T (lines 18–19). Equation (3) is implemented

by lines 23–32. Given a tree $T(v, \mathbf{k_x})$ (line 22), the algorithm attempts to find any existing trees, $T(v, \mathbf{k_y})$, that are also rooted at v and contain keywords $\mathbf{k_x} \cup \mathbf{k_y}$ with more nodes than $T(v, \mathbf{k_x}) + T(v, \mathbf{k_y})$, where $\mathbf{k_x} \neq \mathbf{k_y}$. Any such trees will be replaced with $T(v, \mathbf{k_x}) + T(v, \mathbf{k_y})$ in Q_T (lines 24–28). If there are no such trees, $T(v, \mathbf{k_x}) + T(v, \mathbf{k_y})$ will be enqueued into Q_T (lines 29–30).

We now analyze the worst-case scenario complexity of Algorithm 1 when answering a query Q with a set of keywords $\mathbf{K} = \{k_1, \ldots, k_l\}$ on a data graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. Let $T(v, \mathbf{k})$ be the tree with the minimum number of nodes of all trees rooted at v containing a subset of keywords $\mathbf{k} \subseteq \mathbf{K}$. There are 3 major components in complexity of Algorithm 1: queue maintenance, tree growth and tree merging.

Algorithm 1: Answer Keyword Query Q

Input: $G(V, E)$, $\mathbf{K} = \{k_1, k_2, \ldots, k_l\}$
Output: minimum group Steiner tree $T(v, \mathbf{K})$, $v \in V$

```
 1: Q_T ← Ø;
 2: for each v ∈ V do
 3:     if v contains k ⊆ K
 4:         enqueue T(v, k) into Q_T;
 5:     end if
 6: end for
 7: while Q_T ≠ Ø do
 8:     dequeue Q_T to T(v, k);
 9:     if k = K
10:         return T(v, k);
11:     end if
12:     for each u ∈ N(v) do                          ┐
13:         if ∃ T(u, k) ∈ Q_T                        │
14:             if |T(v, k) + u| < |T(u, k)|          │
15:                 T(u, k) ← T(v, k) + u;            │ tree growth
16:                 Q_T ← T(u, k);                    │ Eq. (2)
17:             end if                                │
18:         else                                      │
19:             T(u, k) ← T(v, k) + u;                │
20:         end if                                    ┘
21:     end for
22:     k_x ← k;
23:     for each k_y s.t. k_x ∩ k_y = Ø do             ┐
24:         if ∃ T(v, k_x ∪ k_y) ∈ Q_T                │
25:             if |T(v, k_x) + T(v, k_y)| < |T(v, k_x ∪ k_y)|
26:                 T(v, k_x ∪ k_y) ← T(v, k_x) + T(v, k_y);  │ tree merging
27:                 Q_T ← T(v, k_x ∪ k_y);            │ Eq. (3)
28:             end if                                │
29:         else                                      │
30:             Q_T ← T(v, k_x) + T(v, k_y);          │
31:         end if                                    ┘
32:     end for
33: end while
```

Queue maintenance. In total, there are 2^l subsets of $\mathbf{K}$. Thus, the maximum length of Q_T is $2^l n$, i.e., every tree rooted at any $v \in V$ containing any $\mathbf{k} \subseteq \mathbf{k}$ is enqueued into Q_T. The complexity of enqueue/update operations and dequeue operations is dependent on the type of the queue. Here, we employ Fibonacci Heap, which has the complexity of $O(1)$ for the enquene/update operations and $O(\log 2^l n)$ for dequeue operations. Because Algorithm 1 will enqueue or dequeue any $T(v, \mathbf{k})$ into/from Q_T at most once, the complexity of enqueuing and dequeuing all $2^l n$ trees in Q_T is $O(2^l n(l + \log n))$.

Tree growth. Lines 12–21 handle the tree growth operations implementing Eq. (2). The **for** loop iterates for $|N(v)|$ times, trying to find the $T(u, \mathbf{k})$ grown from

$T(v, \mathbf{k}) + u$ with the minimum number of nodes. Here, $|N(v)|$ is the total number of neighbors of v. Thus, the total time for Algorithm 1 to execute the comparison operations in lines 12–21 is $O\left(2^l \sum_{v \in V} |N(v)|\right) = O\left(2^l m\right)$.

Tree merging. Lines 23–32 handle the tree merging operations implementing Eq. (3). For each $T(v, \mathbf{k_x})$ dequeued in line 8, the **for** loop in lines 23–32 enumerates every $\mathbf{k_y}$ that fulfils $\mathbf{k_x} \cap \mathbf{k_y} = \emptyset$, where $\mathbf{k_x}, \mathbf{k_y} \subseteq \mathbf{k}$. Given $|\mathbf{K}| = l$, the total number of possible $\mathbf{k_y}$ is $2^{l-|\boldsymbol{kx}|}$. Thus, the total time for Algorithm 1 to execute the comparison operations in lines 23–32 is $n \sum_{i=1}^{l-1} C_{l,i} \times 2^{l-i} = O\left(3^l n\right)$.

Overall, the complexity of Algorithm 1 is $O(2^l n(l + \log n) + 2^l m + 3^l n)$. This indicates that the efficiency of Algorithm 1 relies exponentially on the number of query keywords. In real world problems where l is a small constant, the complexity of Algorithm 1 becomes $O(n\log n + m)$.

3 Experimental Evaluation

We conducted a series of experiments with a prototype of KS3+ implemented using JDK1.6.0 to compare the efficiency (computational overhead) and effectiveness (success rate) of KS3+ with KS3.

3.1 Experimental Setup

The data graphs and queries used in the experiments are randomly generated using a publicly available and widely used dataset named QWS, which contains the functional information about over 2,500 real-world web services [5]. All experiments were conducted on a machine with Intel i5-4570 CPU 3.20 GHz and 8 GB RAM, running Windows 7 ×64 Enterprise. In the experiments, random data graphs are generated based on the Erdős–Rényi model [6]. The relevance between the query keywords determines whether bridging nodes are needed to identify a system solution. In the data graph, directly relevant keywords are composable and hence belong to adjacent nodes. Bridging services are needed when two keywords are not directly relevant. In the experiments, we used the *keyword distance* to represent the relevance between two query keywords, reflected by the number of hops they are away from each other in the data graph. In the experiments, we fixed the keyword distances at 2 for all queries, which were also randomly generated. To avoid very large solutions, we limited the maximum number of nodes to be included in a solution to twice the number of query keywords.

To comprehensively study the impacts of different parameters on the efficiency and effectiveness of KS3+, we vary four parameters in the experiments, as presented in Table 1. Note that in experiment set #3, the number of edges increases with the number of nodes to maintain the graph density while changing the graph size. For each set of experiments, we average the results obtained from 100 runs.

Table 1. Experiment configuration

Parameter	Set #1	Set #2	Set #3	Set #4
Keyword distance	1 to 10	2	2	2
Number of query keywords (l)	2	2 to 6	2	2
Graph size (number of nodes)	2,000	2,000	2,000 to 20,000	2,000
Graph density (number of edges)	2,000	2,000	2,000 to 20,000	2,000 to 8,000

3.2 Evaluation Results

Efficiency. Figure 1 shows the computation times taken by KS3+ and KS3 to answer keyword queries for systems solutions under different parameter settings. Overall, KS3 + demonstrates a multiple orders of magnitude advantage in efficiency over KS3 under different parameter settings. While KS3 often takes seconds to minutes to answer queries under different parameter settings, KS3+ takes less than 1 ms in most cases. This demonstrates its significant advantage in efficiency over KS3.

Figure 1(a) shows the efficiency of KS3+ in identifying the bridging nodes when the keywords in a query are not directly relevant. When the keyword distance increases from 1 to 10, the average computation time of KS3 increases from 16 ms to 2,899 ms. In the meantime, the average computation time of KS3+ increases from 0.08 ms to 0.40 ms. KS3+ outperforms KS3 significantly, and demonstrates much higher tolerance to the increase in keyword distance. The results shown in Fig. 1(a) demonstrate that KS3+ can efficiently find a system solution even if the keywords entered are only remotely relevant, thanks to its excellent ability to identify bridging nodes.

Figure 1(b) demonstrates the outstanding ability of KS3+ to find a system solution when multiple bridging nodes are needed to connect many keyword nodes. KS3+ demonstrates great performance with an increase from 0.42 ms to 319.69 ms in computation time in response to the increase in the number of query keywords (**referred to as *l* hereafter**) from 2 to 5. The corresponding increase in the computation time of KS3 is from 1,645 ms to 12,574 ms. Again, KS3+ outperforms KS3 significantly. In particular, when l reaches 6, it takes KS3+ 2,777.92 ms on average to find a system solution, while KS3 cannot even answer the query within a reasonable amount of time. That is why the corresponding data is missing for KS3 in Fig. 1(b). Figure 1(b) shows that KS3+ has a considerably better ability to find bridging nodes than KS3.

Figure 1(c) shows that the increase in the computation time of KS3 increases rapidly with the graph size, while the increase in the computation time KS3+ is almost negligible. On a very large data graph with 20,000 nodes, KS3 takes a significant amount of time (up to 75,000 ms) to answer a query. In the meantime, KS3+ takes only 1.35 ms on average to answer the same query. In a large data graph, the number of group Steiner trees that cover all the keyword nodes is extremely large even when the number of keywords to cover is small. KS3 needs to identify and inspect all those trees. The extremely large search space inevitably leads to long computation time of KS3. KS3+, on the other hand, does not have to inspect all those trees. It prunes invalid trees and grows or merges only the trees that are likely to be part of the final answer tree. Thus, KS3+ can handle queries over large data graphs much more efficiently than KS3.

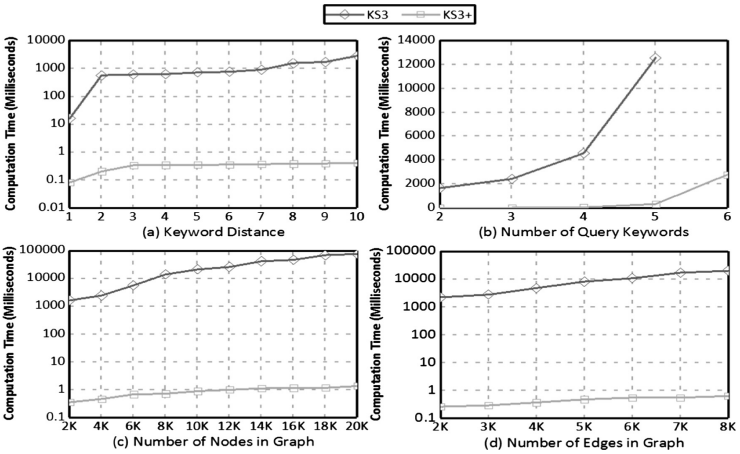

Fig. 1. Computation time under different parameter settings (keyword distance = 2)

Figure 1(d) shows that in a dense data graph, where each service has many neighbors, it takes KS3+ much less time than KS3 to find a system solution. The advantage of KS3+ over KS3 is by multiple orders of magnitudes, similar to the results shown in Fig. 1(a) and (c). As the number of edges increases from 2,000 to 8,000, the average computation time of KS3+ increases accordingly from 0.27 ms to 0.64 ms, versus the increase from 2,256 ms to 20,331 ms for KS3. A higher graph density means more neighbors for each node, leading to more group Steiner trees for KS3 to identify and inspect to answer a query. However, given a tree $T(v, \mathbf{k})$ dequeued in line 8 of Algorithm 1, out of all the neighbors of v, Algorithm 1 would only grow $T(v, \mathbf{k})$ to include those that result in trees containing the same keywords as $T(v, \mathbf{k})$ but with fewer nodes. This prunes most invalid trees and ensures the high efficiency of KS3+.

Effectiveness. We compared the effectiveness of KS3+ and KS3, measured by success rate, i.e., the percentage of cases where an answer to the keyword query can be found. Overall, KS3+ is as effective as KS3, with a consistent success rate of 100% in all experiments under different parameter settings. This indicates that KS3+ can always find a system solution, like KS3. The experimental results demonstrate that KS3+ does not compromise the success rate in finding a solution.

4 Related Work

The process for building an SBS consists of three phases: system planning, service discovery and service selection.

System planning. The system engineer identifies the system tasks required for the target SBS, as well as their execution order. Most system planning techniques are based on artificial intelligence techniques [7]. The general idea is to model the task identification

problem as a planning problem. For example, in [7], the authors model the task identification problem as a CSTE planning problem to be solved with an SCP solver.

Service discovery. Through service registries or service portals, the system engineer identifies a set of candidate services for each of the identified system tasks based on the functional and semantic information of candidate services. To improve the accuracy of service matching, several semantic web service languages have been proposed based on ontology techniques, e.g., OWLS-MX [8]. It automates the service matching operation that identifies the services that can perform the required system tasks. Many approaches have been proposed to automate the service discovery process, based on ontology techniques such as logical reasoning and temporal planning [9].

Service selection. The system engineer selects one service from the candidate services for each system task to compose the target SBS. The selected services must collectively fulfil the multi-dimensional quality constraints for the SBS [4], e.g., reliability, response time, cost, etc., which is an NP-complete problem. Integer Programming (IP) is the main technique adopted in this phase. AgFlow [2] is one of the most representative approaches. Following the idea of AgFlow, many researchers have been trying to reduce the computation time for quality-aware service selection [10] or to solve the problem in more complex environments [1, 11].

A planning technique was proposed that explores system solutions by looking up services whose tags match the tags describing the SBS [3]. For each query, the engineer needs to enter a source tag and a destination tag. The proposed technique heuristically identifies the possible service compositions with an entry service according to the source tag and an exit service according to the destination tag. A similar approach is proposed in [12]. A major limitation to these approaches is that each query allows only two tags, i.e., a source tag and a destination tag. Multiple tags can only be entered one by one in different queries that are processed individually until a final solution is found. An error made in an early query can easily make it impossible to find the final solution.

KS3 was proposed in [4]. It overcomes the limitations of the approaches proposed in [3, 12]. However, it suffers from extremely poor efficiency in large-scale scenarios. By modelling keyword queries as dynamic programming problems, KS3+ achieves significantly higher efficiency without sacrificing effectiveness.

5 Conclusions and Future Work

In this paper, we propose KS3+, a novel approach that integrates and automates the system planning, service discovery and service selection operations for building service-based systems (SBSs). It assists system engineers without detailed knowledge of SOA techniques in finding system solutions with only a few keywords that describe the required system tasks. KS3+ offers a new paradigm for building SBSs and can significantly save the time and effort during the process for building SBSs. Making no compromise in effectiveness, KS3+ significantly outperforms KS3 in efficiency.

In our future work, we will enhance KS3+ to answer queries with quality constraints and quality optimization goals.

Acknowledgment. This work is partly supported by Australian Research Council Projects DP170101932, DP150101775 and LP130100324.

References

1. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. **33**(6), 369–384 (2007)
2. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)
3. Liu, X., Ma, Y., Huang, G., Zhao, J., Mei, H., Liu, Y.: Data-driven composition for service-oriented situational web applications. IEEE Trans. Serv. Comput. **8**(1), 2–16 (2015)
4. He, Q., Zhou, R., Zhang, X., Wang, Y., Ye, D., Chen, F., Grundy, J., Yang, Y.: Keyword search for building service-based systems. IEEE Trans. Softw. Eng. **437**(7), 658–674 (2016)
5. Al-Masri, E., Mahmoud, Q.H.: Investigating web services on the world wide web. In: 17th International Conference on World Wide Web (WWW 2008), pp. 795–804 (2008)
6. Durrett, R.: Random Graph Dynamics. Cambridge University Press, Cambridge (2007)
7. Zou, G., Lu, Q., Chen, Y., Huang, R., Xu, Y., Xiang, Y.: QoS-aware dynamic composition of web services using numerical temporal planning. IEEE Trans. Serv. Comput. **7**(1), 18–31 (2014)
8. Klusch, M., Fries, B., Sycara, K.P.: OWLS-MX: a hybrid semantic web service matchmaker for OWL-S services. J. Web Semant. **7**(2), 915–922 (2009)
9. Cassar, G., Barnaghi, P., Moessner, K.: Probabilistic matchmaking methods for automated service discovery. IEEE Trans. Serv. Comput. **7**(4), 654–666 (2014)
10. Trummer, I., Faltings, B., Binder, W.: Multi-objective quality-driven service selection - a fully polynomial time approximation scheme. IEEE Trans. Softw. Eng. **40**(2), 167–191 (2014)
11. He, Q., Yan, J., Jin, H., Yang, Y.: Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. IEEE Trans. Softw. Eng. **40**(2), 192–215 (2014)
12. Huang, G., Ma, Y., Liu, X., Luo, Y., Lu, X., Blake, M.B.: Model-based automated navigation and composition of complex service mashups. IEEE Trans. Serv. Comput. **8**(3), 494–506 (2015)