User-Centric, Heuristic Optimization of Service Composition in Clouds

Kevin Kofler, Irfan ul Haq, and Erich Schikuta

Department of Knowledge and Business Engineering
University of Vienna, Austria
kevin.kofler@chello.at, {irfan.ul.haq,erich.schikuta}@univie.ac.at

Abstract. With the advent of Cloud computing, there is a high potential for third-party solution providers such as composite service providers, aggregators or resellers to tie together services from different clouds to fulfill the pay-per-use demands of their customers. Customer satisfaction which is primarily based on the fulfillment of user-centric objectives is a crucial success factor to excel in such a service market. The clients' requirements, if they change over time even after the desired solution composition, may result in a failure of this approach. On the other hand, business prospects expand with the possibility of reselling already designed solutions to different customers after the underlying services become available again. The service composition strategies must cope with the above-mentioned dynamic situations.

In this paper we address these challenges in context with the customer-driven service selection. We present a formal approach to map customer requirements onto functional and non-functional attributes of the services. We define a happiness measure to guarantee user satisfaction and devise a parallelizable service composition algorithm to maximize this happiness measure. We devise a heuristic approach based on historical information of service composition to rapidly react to changes in client requirements at design time and indicate run-time remedies such as for service failures. The heuristic algorithm is also useful to recompose similar solutions for different clients with matching requirements. Our algorithms are evaluated by the results of a simulation developed on the workflow tool Kepler coupled with a C++ implementation of the optimization algorithms.

1 Introduction

In this paper, we pursue our vision of *IT-based service markets*, which opens up doors for totally new business processes for *consumers and producers*. So in the near future it will be a common practice to sell IT resources as services and not as goods. *For example, "Writing a letter" can be as simple as using a telephone: Forget buying software and hardware! All we need is a simple interface to the services on the Internet, both the word processor functionality and the necessary physical resources (processor cycles and storage space); and everything is paid transparently via our telephone bill.*

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 405–417, 2010.

Cloud Computing, building on the notions of Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) strives for a similar vision to provide a platform for future free markets of IT resources. Marked as the most popular emerging technology of 2009 on the Gartner Hype Cycle [3], Cloud Computing attracts both service providers and consumers alike with its promise of cost reduction based on the pay-per-use model and the shift from the usual capital upfront investment model to an operational expense [2].

There is a high potential for third-party solution providers such as Composite Service Providers (CSPs) [7], aggregators or resellers to tie together services from different clouds [6] to fulfill the pay-per-use demands of their customers. Several CSPs, if joined together in a hierarchical fashion, will lead to service supply chains with a service composition process taking place at every level of the chain. Such compositions will result in new business processes underpinned by new business networks. The pay-per-use service models will not only help in the reduction of cost but various services, after being composed together with the help of orchestration technology, will become available again as more capable composite services with a guaranteed level of service.

Customer satisfaction, which is primarily based on the fulfillment of user-centric objectives, is a crucial success factor to excel in such a service market[4]. Consumers will be able to access services from the Cloud under their desired level of service by mentioning the Quality of Service (QoS) requirements. Consumers and service providers will be bound to these requirements through Service Level Agreements (SLAs).

The most convenient way for the end-user to specify its requirements is to first represent them in form of an abstract workflow, which allows a user to draw a sequence of activities representing his desired services along with the user's functional and nonfunctional requirements. In the next step, the properties of the required services can be expressed in form of SLA templates in a registry similar to [5] where both user requirements and service offerings are represented as SLA templates so that both parties can discover each other. Suitable services satisfying user requirements are then searched from the registry, and mapped on the abstract workflow. The optimal service composition requires the best selection out of the available services. The selection of the services is based on user requirements. These requirements can be functional requirements such as order of the required services or non-functional such as total cost, total response time, availability, reliability and trust etc. Even after a service composition, there can be a need of recomposition, for reasons such as the user changing his requirements or service failures. This is a multi-fold problem and poses several challenges. Automatic adjustment and recovery in such situations demand a strong formal model to realize various interrelated entities contributing to service compositions, such as: the formal and non formal attributes of services, user requirements, and criteria for the service selection. The next challenge requires optimization algorithms to ensure the best design and performance of the composed services. There must be fault tolerance mechanisms that deal with the runtime failure of services. In this paper we present:

- a formal model to automate the selection of optimal services fulfilling user requirements
- a heuristic algorithm to cope with dynamic changes in user requirements by updating an existing solution
- the design of our simulation environment, incorporating the Kepler workflow tool [18], and C++ based optimization components, and
- the implementation details and a performance analysis of our model

In the formal model, the user requirements are transformed to the QoS parameters of the services. We define a *happiness measure* based on these QoS parameters to grade the set of candidate services and as a result, the one with the highest happiness measure is selected. The heuristic algorithm keeps track of the best services and utilizes this historical information to remodel the service composition in response to the changing user requirements. In this paper, we have given a special focus to the cost and time requirements but our approach is equally valid for other types of functional and non functional attributes.

This paper complements our previous work in this area, where we employ the data access mechanism of the Datagrid project of CERN to define a cost model for data replication [13], design a blackboard approach for dynamic workflow optimization [9] and devise a formal model and a parallelized algorithm for user-centric service compositions [8]. The typical user requirements in the Datagrid project include the bandwidth, disk space, compression algorithms, criteria of replication and the preferred sites for data storage. Our two-phase heuristic algorithm for workflow optimization has been designed to coordinate with our blackboard approach and the cost model.

In section 2, we present our formal model for QoS optimization of service composition. Section 3 elaborates the two-phase heuristic algorithm for QoS optimization of service composition, whereas the sections 4 and 5 discuss the implementation details and the performance analysis respectively. Section 6 states the related work and section 7 winds up the paper with the conclusion and future work.

2 QoS Optimization of Workflows

2.1 Motivational Scenario

We start with a scenario. Our user Alice wants to use the computing power of the Cloud for a graphical simulation. The simulation needs to be computed by a physics engine based on a mathematical model provided as input, rendered by a graphical engine using graphics and sound libraries, converted to a compressed video file by a codec – all examples of Software as a Service (SaaS) – and saved to a storage provider providing Infrastructure as a Service (IaaS). She formulates her requirements to a Platform as a Service (PaaS) provider in the Cloud through some interface which is out of the scope of this paper. The PaaS provider which Alice contacted and which appeared to her as a single provider is actually just an aggregator which delegates its activities to a physics engine service, a graphical

engine service (which may itself be composite, using graphics and sound library services), a codec service and a storage service in form of a workflow. Thus, it builds an abstract workflow of the above four activities and maps the requirements Alice has described onto those tasks. Some of those requirements *must* be fulfilled strictly, for instance she might require a maximum error of 0.1% in the physics, whereas others such as the desired compression rate of the output video *should* be satisfied as well as possible but are rather flexible. The aggregator then runs a branch and bound algorithm to search for the services that can map onto different tasks of the workflow, automatically selecting the best services fulfilling Alice's requirements. A codec service out of many competing peers is selected on the basis of the lowest cost and high degree of compression. Later on, unsatisfied by the quality of output, Alice decides to change some requirements and increases the maximum allowed cost. The automated workflow tool uses a heuristic algorithm to cope with the new user preferences.

A few hours later, a new user Bob also wishes to run a similar simulation using the same composite service provider. His requirements are very similar to Alice's, so instead of running a branch and bound algorithm from scratch, the provider heuristically updates the solution computed for Alice to match Bob's requirements.

2.2 Formal Model for QoS Optimization of Workflows

The given scenario poses a complex mathematical problem with interrelated issues. We formally define and summarize the relationships among these issues. A more detailed, fully formalized description of our model can be found in [8]. In our model, we denote the set of equivalent services, e.g. the video compression services, as a *service class*. We assume that each service belongs to exactly one service class. We model QoS parameters such as the compression rate as numeric *service attributes*, common to all services of a given class, but with different numeric *values* Q_0 for the different offered services. For simplicity, we will assume values in real numbers, i.e. in the set $\mathbb{R}$, this also covers booleans if we identify *true* as 1 and *false* as 0. We *normalize* those values to qualities Q in the range $[0, 1]$, where 0 is the worst possible quality and 1 the best one. The map used for this normalization is the same for all services of a given class, making the normalized values comparable. It can be increasing (for attributes which directly indicate a quality) or decreasing (for attributes such as latencies where less is better).

An *abstract workflow* W_0 specifies the requirements the user has for the workflow. It is given by a directed graph whose nodes are the steps in the workflow, and for each node $v \in V$ (the set of workflow nodes), the needed service class $f_0(v)$, minimum ("must") and desired ("should") requirements for the attributes, which are respectively normalized to vectors $R_m(v) \leq R_d(v)$ (componentwise) of numbers in $[0, 1]$ using the same map as for the attribute values, and weights $w(v)$ given to the desired requirements, indicating how much value is given to the "should" request. We will not use the graph structure in our algorithm, thus we only consider a list of nodes with the above properties.

A *concrete workflow* W specifies a concrete workflow instance which should match the user's requirements. We define a concrete workflow as a list of nodes v with a requested service $f(v)$.
We say W is *sensible* for W_0 if they have the same number of nodes (or as graphs, the same graph structure) and the service for each node is actually of the requested class for that node.
We say W is *feasible* for W_0 if W is sensible for W_0 and $\forall v \in V : Q_{f(v)} \geq R_m(v)$ (componentwise), i.e. if all the minimum requirements are satisfied.

We define a *happiness measure* which quantifies how happy the user is with a given workflow considering his/her desired requirements and weights. For all pairs (W_0, W) where W is sensible for W_0, we define $h(W_0, W)$ as

$$h(W_0, W) = \sum_{v \in V} h_{vf_0(v)}$$

where for each choice of service s for v

$$h_{vs} = \sum_{l=1}^{m_s} w_l(v) h_{vsl}$$

(m_s being the number of attributes of the service s) and

$$h_{vsl} = \begin{cases} 0, (Q_s)_l < (R_m(v))_l \\ 1, (Q_s)_l \geq (R_d(v))_l \\ \frac{(Q_s)_l - (R_m(v))_l}{(R_d(v))_l - (R_m(v))_l}, \text{else} \end{cases},$$

i.e. 0 for infeasible qualities, 1 for qualities at least as high as desired and linearly increasing between the minimum and the desired requirement. Note that this is a linear happiness measure. We assume it makes sense to define such a linear happiness, which is a requirement on the normalization maps.

With the above definitions, we can state the problem in mathematical terms: Given an abstract workflow W_0, we want to find a concrete workflow W which optimizes:

$$\begin{aligned} &\max h(W_0, W) \\ &\text{s.t. } W \text{ is feasible for } W_0 \\ &\quad \textstyle\sum_{v \in V} \left(Q_{0_{f(v)}}\right)_1 \leq K \\ &\quad \textstyle\sum_{v \in V} \left(Q_{0_{f(v)}}\right)_2 \leq T \end{aligned}$$

where $Q_{0_{f(v)_1}}$ is assumed to be the *cost* of the service $f(v)$ and $Q_{0_{f(v)_2}}$ its *execution time*, and thus K and T are upper bounds on the total cost and the total execution time, respectively. We will call those additional constraints *aggregate constraints*, because they are the constraints which aggregate the QoS parameters of the different services, whereas workflow feasibility considers each node individually. Our approach can also handle other similar constraints.

It can be easily seen that, after filtering out the services which do not satisfy the minimum requirements, our problem becomes equivalent to a Multidimensional Multi-choice Knapsack Problem (MMKP) [1]: the utilities in the MMKP are our happiness values h_{vs}.

3 A Heuristic Algorithm for Optimization of Service Composition

In this section, we will present an algorithm to realize our motivational scenario in two phases: a precomputation phase in which we aim at the QoS-aware optimization of service composition and an updating phase using heuristics to react to dynamic changes in user requirements or reuse solutions for users with similar requirements.

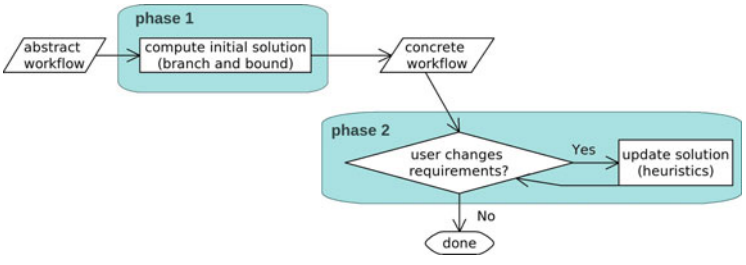

Fig. 1. The two phases of the optimization algorithm

3.1 Phase 1: Precomputation

For the precomputation phase, we use a branch and bound algorithm, one of the most successful, flexible and easy to parallelize algorithms for optimization problems. The input data for our algorithm is composed of a vector $V = \{v_1, \ldots, v_p\}$ of nodes in the workflow graph and a vector S of vectors $S_i = \{s_{i1}, \ldots, s_{ik_i}\}$ of services (one vector S_i for each service class). The entries in the vectors are simple structures: a node has an integer f_0, the *service class* required for the node in the abstract workflow and vectors R_m, R_d, w, the *normalized requirements* and *weights* for each attribute of the service class; a service has vectors Q, its *normalized QoS parameters* and Q_0, its *actual QoS parameters* (used for the aggregate constraints). Our solutions are represented as *decision vectors* $d = \{d_1, \ldots, d_p\}$, each entry d_v of which corresponds to the choice of a service for the node v. The details of our branch and bound approach are out of the scope of this paper, they can be found in [8].

3.2 Phase 2: Dynamic Heuristic Updates

The drawback of the branch and bound approach is that it is expensive. Its worst-case performance can be proven to be exponential. Its practical performance is highly dependent on input data, and in the case of a threaded or distributed implementation, also non-deterministic. (The algorithm is non-deterministic due to thread or process scheduling, in the distributed case also timing of network communication. Only the resulting happiness is deterministic, as the algorithm is guaranteed to find the exact optimum.) Unfortunately, this is not merely an issue with the implementation: the problem is NP-hard, due to the total cost

and time constraints, which make it equivalent to a two-dimensional knapsack problem. One of the ways to deal with this scalability problem is to parallelize the implementation. In [8], we have achieved promising results with a CORBA-based distributed implementation, showing nearly linear speedup on sufficiently large testcases, however that is out of the scope of this paper. We will instead focus on techniques fast enough to react to changes to the problem in real time.

As explained above, in order to efficiently react to changed user requirements, a different approach is needed, based on heuristics. Our proposed solution is to reuse the existing solution to the original problem as a starting point and update it for the changed requirements. This can be done efficiently (in low-order polynomial time) and generally results in a near-optimal solution to the modified problem. However, this is only a heuristic: the optimality of the obtained solution cannot be guaranteed. Only recomputing everything, which is NP-hard (as discussed above), can guarantee that.

Algorithm 1 describes the approach used for the total cost constraint. Exactly the same procedure is also used for the total time constraint, and in principle similar updates (whose details are beyond the scope of this paper) could also be done for other changes in user requirements or service offerings, e.g.:

- changes in user's quality requirements or weights
- changes in service parameters
- added / removed services.

As this procedure is very efficient, it is also be possible to use it for runtime changes, e.g. service failure. In this case, we have to consider the structure of the workflow, as it does not make sense to change a service which already completed or with which a non-refundable SLA has already been agreed to. Thus, one would have to replace the failed service with another service, then make up for cost or time overruns, if any, by replacing the services which are not fixed yet by cheaper resp. faster services.

The heuristic update can also be employed to reuse a solution designed for a given client for a new client with similar requirements. This case can be treated just as if the original client changed their requirements.

Another possibility worth trying would be to avoid the branch and bound procedure entirely and rely only on the updating heuristics:

1. We compute a solution without the constraints on K and T. This can be done in polynomial time. We use this solution as our starting solution.
2. We update that solution heuristically to honor K and T.

This procedure would be significantly faster than branch and bound, but the drawback is that it would no longer be guaranteed to find an optimal solution. Another issue is that the current heuristics may fail entirely, because adjusting for the new value of one constraint can violate the other. Thus, we implemented a *failsafe* version of algorithm 1 which considers only those alternative services which do not take more time when adjusting for cost and vice-versa. We try algorithm 1 first, then if it fails, the failsafe version. When adjusting for an

```
1  Compute cost of old solution;
2  if K > old K then
3  |  foreach node do
4  |  |  Compute happiness for currently chosen service;
5  |  |  foreach service satisfying minimum requirements, happiness
   |  |  hs > old hs and cost > old cost do
6  |  |  |  Compute q = (hs − old hs) / (cost − old cost);
7  |  |  end
8  |  |  Add best (largest) q to priority queue;
9  |  end
10 |  while queue not empty do
11 |  |  Pick top queue entry;
12 |  |  if new cost ≤ K then apply update;
13 |  end
14 |  Check solution;
15 |  if infeasible then return FAILURE;
16 |  Output updated solution;
17 else if cost > K then
18 |  foreach node do
19 |  |  Compute happiness for currently chosen service;
20 |  |  foreach service satisfying minimum requirements, happiness
   |  |  hs < old hs and cost < old cost do
21 |  |  |  Compute q = (hs − old hs) / (cost − old cost);
22 |  |  end
23 |  |  Add best (smallest) q to priority queue;
24 |  end
25 |  while queue not empty and cost > K do
26 |  |  Pick top queue entry;
27 |  |  Apply update;
28 |  end
29 |  Check solution;
30 |  if infeasible then return FAILURE;
31 |  Output updated solution;
32 else
33 |  Output old solution;
34 end
35 return SUCCESS;
```

Algorithm 1. Heuristic update for K

increased constraint, this will always lead to a feasible solution which is at least as good as the initial one. For a decreased constraint, it can still fail, in which case our implementation falls back to recomputing a new solution using branch and bound. A completely failsafe approach to maintaining feasibility is NP-hard, just like the original problem, because it amounts to minimizing one constraint while satisfying the other, which is equivalent to a knapsack problem.

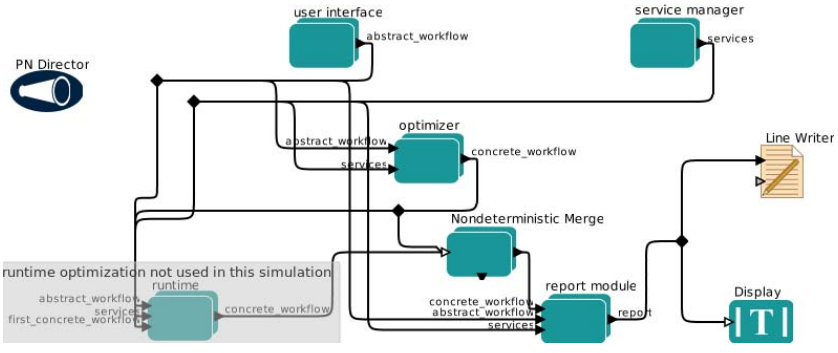

Fig. 2. Structure of our workflow (screenshot from Kepler)

4 Implementation

To model the different components of our optimizer, we used the workflow tool Kepler [18]. However, there is no easy way to represent a parallel branch and bound algorithm in Kepler, as it can only spawn a fixed amount of threads or processes (one for each component in the Kepler workflow), whereas we would need a way to spawn them dynamically, one for each search graph node. Other workflow tools, such as Triana and Taverna, have this same limitation. Thus, we decided to use an external process for the optimizer core, cooperating with the Kepler workflow. The core is written in C++ using `libQtCore` [19].

Figure 2 shows the workflow used for our optimization method: there are components for the user interface, a service manager, the optimizer and a report module. In the simulation, we use simple dialogs or files for input. In the real world, we would have a user-friendly graphical user interface and the service manager would communicate with third-party services. Our design also allows for a runtime optimization module which can react to dynamic changes at runtime, but that component is out of the scope of this paper.

The first implementation of the optimizer core was based on threads. As threads have high overhead and as they mean effectively leaving the search strategy to the operating system (because the thread scheduler decides which node in the search graph to process next), which knows nothing about the structure of our problem, we also implemented a deterministic sequential version using breadth-first search.

5 Performance Analysis

We compared our branch and bound implementation with the updating heuristics to see how much faster the heuristics are.

5.1 Test Method

As we did not have access to any real-world data, we generated some feasible, but otherwise random, synthetic testcases with a pseudorandom number generator.

We did tests with 10, 12, 15, 20, 25 and 30 workflow nodes, each of which was mapped to a different service class. The timings reported for the branch and bound algorithm are from a sequential breadth-first implementation, which proved the most efficient on a single CPU.

For each testcase, we ran a series of tests with constant services and user requirements, changing only the values of K and T (one at a time) and we compared the results of the heuristics with repeated runs of branch and bound, looking at both the execution time of the optimization process and the quality of the solution (i.e. how close to the optimum it is).

The tests were run on a single-core 2.6 GHz Pentium 4 with HyperThreading disabled.

We used the following pairs (K, T) for each testcase:

10 nodes: $(350, 500)$, $(400, 500)$, $(400, 550)$, $(400, 500)$, $(380, 500)$, $(350, 500)$,

12 nodes: $(400, 600)$, $(450, 600)$, $(450, 650)$, $(450, 600)$, $(430, 600)$, $(400, 600)$,

15 nodes: $(550, 700)$, $(600, 700)$, $(600, 750)$, $(600, 700)$, $(580, 700)$, $(550, 700)$,

20 nodes: $(800, 850)$, $(850, 850)$, $(850, 900)$, $(850, 850)$, $(830, 850)$, $(800, 850)$,

25 nodes: $(900, 1200)$, $(950, 1200)$, $(950, 1250)$, $(950, 1200)$, $(930, 1200)$, $(900, 1200)$,

30 nodes: $(950, 1500)$, $(1000, 1500)$, $(980, 1500)$, $(950, 1500)$.

5.2 Results

Figure 3 shows the results of the performance measurements. Almost invisible bars in the figures mean the value is very small or zero. The results show that the branch and bound algorithm scales up to problem sizes in the order of 30 workflow nodes with acceptable performance, but that the heuristic update is several orders of magnitude faster. Note that the first testcase of each set is the initial solution, which is always computed using branch and bound.

We also found that in the testcases with 10 and 12 workflow nodes, the heuristic updates always found the optimum solution. This is not always guaranteed, because the update is only a heuristic. Indeed, for the testcases with 15 or more workflow nodes, the solutions found by the heuristic approach were not always optimal, but they came very close (within 98% of the happiness) to the optimum. Figure 4 shows the ratios between the happiness values for the solutions found by the heuristics and the optimum happiness values as found by branch and bound.

We also tried using only the heuristics instead of the branch and bound process, using the solution without constraints for K and T as the starting solution. This turned out to be much faster than branch and bound, which matches the expectations, as the heuristics are polynomial, whereas the branch and bound is exponential. The initial heuristic updates are as fast as the subsequent ones. Unfortunately, this is only preliminary due to the problems described in the previous section: the heuristic update can fail if the update to reduce the value of one constraint makes it exceed the bound for the other one, the "failsafe" version can fail if a more expensive service needs to be picked to reduce the computation time or vice-versa, and there is no guarantee of optimality. However, these problems are inherent to heuristics.

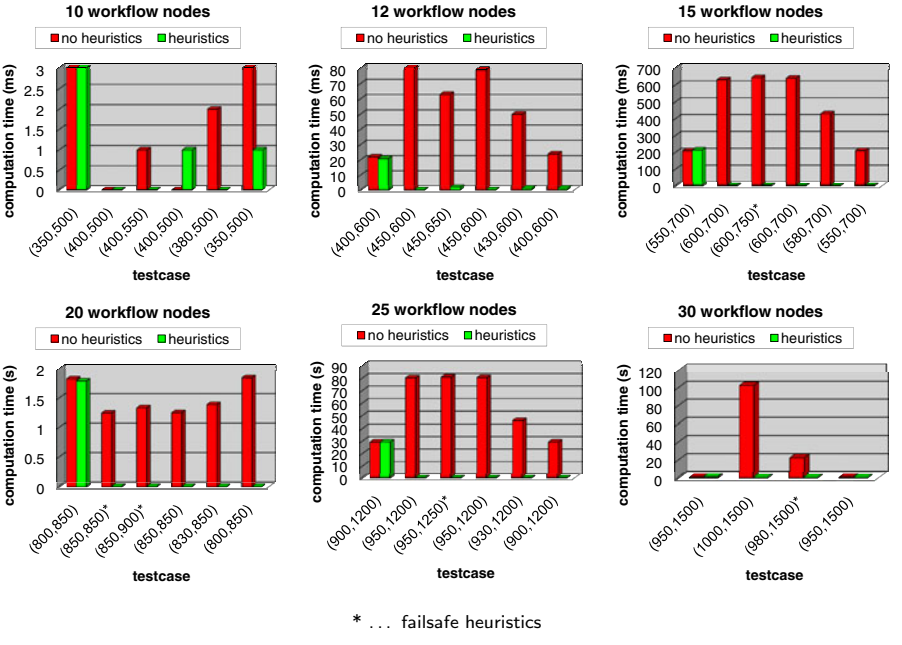

Fig. 3. Performance comparison branch and bound vs. heuristic update

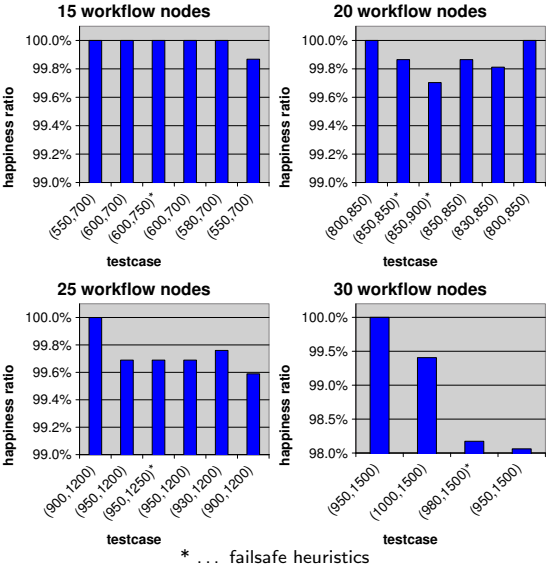

Fig. 4. Happiness ratio between heuristic solution and optimum

We have presented a solution to the well-known problem of user-centric optimization of service composition and our approach shows its qualities in the second phase by demonstrating very promising results for real-time response to changing user requirements. Even more, in practical use the two-phase algorithm as a whole showed an acceptable runtime behavior, justifying it to be a working solution to the workflow optimization problem.

6 State of the Art

The expert group report EC Information Society and Media [2], published in 2010, has highlighted the role of resellers and aggregators in the Cloud Computing, who will aggregate resources according to the customer requirements. Buyya et al. [6] have also described the third-party resellers who can tie together services from different Clouds to meet the customer-defined QoS constraints. QoS constraints are an essential part of the workflow design and play an important role in service selection and scheduling [10] [14]. Binder et al. [15] extract the resource requirements of the services from the OWL-S [17] descriptions that allow defining nonfunctional properties of the components. A mathematical model then computes the execution cost of the workflow and a genetic algorithm is used afterwards to optimize the workflow execution. This approach very successfully maps the resources on workflow tasks but does not discuss dynamically changing conditions. Huang et al. [16] present a very good approach to workflow optimization by dynamic web service selection. An optimal service is selected based on the history data and real-time data. Their approach does not discuss the case of adapting to user-defined QoS constraints. Jia Yu et al. [10] propose a QoS-based workflow management system and scheduling algorithm for the service Grid that minimizes the execution cost and yet meets the time constraints imposed by the user. The QoS-level constraints can be defined at task level as well as at workflow level. In [11] and [9], we developed a blackboard [12] approach coupled with an A^* algorithm to automatically construct and optimize the Grid-based workflows. We used CERN's Datagrid project as a use case for our approach. The blackboard knowledge sources assess the characteristic parameter values for specific services such as cost, execution speed, bandwidth etc. needed in the construction process. This paper complements our previous work and aims at the challenge of workflow optimization at the next level.

7 Conclusion and Future Work

We have presented a two-phase heuristic algorithm to optimize service composition in Clouds that is based on a well-defined formal model. We have presented the details of its implementation using the workflow tool Kepler coupled with our optimization algorithm written in C++. The computation-intensive initial optimization can be done in parallel, a CORBA-based version of our algorithm demonstrating its scalable behavior in heterogeneous distributed environments is presented in [8]. We have also elaborated our heuristic optimization strategy for efficiently reacting to changing user requirements or reusing solutions

for users with similar requirements and briefly introduced its extension to runtime changes in service availability. We are working on the runtime optimization module that will be integrated with our system.

References

1. Yu, T., Lin, K.-J.: Services Selection Algorithms for Composing Complex Services with Multiple QoS Constraints. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 130–143. Springer, Heidelberg (2005)
2. Jeffery, K., et al.: The Future of Cloud Computing: Opportunities for European Cloud Computing beyond 2010, EC, Information Society and Media (2010)
3. Gartner Hype Cycle for Emerging Technologies (2009), `http://www.gartner.com/it/page.jsp?id=1124212` (last access: February 2010)
4. Van Looy, B., Gemmel, P., Van Dierdonck, R. (eds.): Services Management: An Integrated Approach, in: Financial Times. Prentice Hall, Harlow (2003)
5. Brandic, I., Music, D., Leitner, P., Dustdar, S.: *VieSLAF* Framework: Enabling Adaptive and Versatile SLA-Management. In: Altmann, J., Buyya, R., Rana, O.F. (eds.) GECON 2009. LNCS, vol. 5745, pp. 60–73. Springer, Heidelberg (2009)
6. Buyya, R., et al.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. In: Future Generation Computer Systems, vol. 25, pp. 599–616. Elsevier, Amsterdam (2009)
7. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web services: concepts, architectures and applications. Springer, New York (2004)
8. Kofler, K., Haq, I.U., Schikuta, E.: A Parallel Branch and Bound Algorithm for Workflow QoS Optimization. In: Proc. ICPP 2009, Vienna, Austria (September 2009)
9. Schikuta, E., Wanek, H., Haq, I.U.: Grid Workflow Optimization regarding Dynamically Changing Resources and Conditions. Journal of CCPE (2008)
10. Yu, J., Buyya, R., Tham, C.-K.: QoS-based Scheduling of Workflow Applications on Service Grids, in Technical Report, GRIDS-TR-2005-8, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, June 9 (2005)
11. Wanek, H., Schikuta, E.: Using Blackboards to Optimize Grid Workflows with Respect to Quality Constraints. In: GCCW, Huhan, China (October 2006)
12. Corkill, D.D.: Blackboard systems. AI Expert 6(9), 40–47 (1991)
13. Stockinger, H., Stockinger, K., Schikuta, E., Willers, I.: Towards a Cost Model for Distributed and Replicated Data Stores. In: Proceedings of 9th Euromicro Workshop on Parallel and Distributed Processing (PDP 2001). IEEE CS Press, Los Alamitos (2001)
14. Chen, J., Yang, Y.: Adaptive Selection of Necessary and Sufficient Checkpoints for Dynamic Verification of Temporal Constraints in Grid Workflow Systems. ACM Transactions on Autonomous and Adaptive Systems, Article 6 2(2) (June 2007)
15. Binder, W., Constantinescu, I., Faltings, B., Heterd, N.: Optimal Workflow Execution in Grid Environments. In: NODe/GSEM 2005, pp. 276–295 (2005)
16. Huang, L., Walker, D.W., Huang, Y., Rana, O.F.: Dynamic Web Service Selection for Workflow Optimisation. In: Proceedings of 4th UK e-Science Programme All Hands Meeting (AHM), Nottingham, UK (September 2005)
17. Martin, D., et al.: Bringing Semantics to Web Services: The OWL-S Approach. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 26–42. Springer, Heidelberg (2005)
18. The Kepler workflow tool, `https://kepler-project.org/` (last access June 2010)
19. Qt – A cross-platform application and UI framework, `http://qt.nokia.com/` (last access June 2010)