Automated QoS-Aware Service Selection Based on Soft Constraints

Elias Keis[1,2,3], Carlos Gustavo Lopez Pombo[4], Agustín Eloy Martinez Suñé[5(✉)], and Alexander Knapp[1]

[1] Universität Augsburg, Augsburg, Germany
elias.keis@tum.de, alexander.knapp@uni-a.de
[2] Technische Universität München, Munich, Germany
[3] Ludwig-Maximilians-Universität München, Munich, Germany
[4] Universidad Nacional de Río Negro and CONICET, San Carlos de Bariloche, Argentina
cglopezpombo@unrn.edu.ar
[5] Universidad de Buenos Aires and CONICET, Buenos Aires, Argentina
aemartinez@dc.uba.ar

Abstract. QoS attributes are one of the key factors taken into account when selecting services for a composite application. While there are systems for automated service selection based on QoS constraints, most of them are very limited in the preferences the user can state. In this paper we present: a) a simple, yet versatile, language for describing composite applications, b) a rich set of notations for stating complex preferences over the QoS attributes, including checkpoints and invariants, and c) an automatic tool for optimal global QoS-aware service selection based on MiniBrass, a state-of-the-art soft-constraint solver. We provide a running example accompanying the definitions and a preliminary performance analysis showing the practical usefulness of the tools.

Keywords: Service selection · Soft-constraint solving · Quality of service · Service-oriented computing

1 Introduction

In software-as-a-service paradigms such as service-oriented computing, software systems are no longer monolithic chunks of code executing within the boundaries of an organization. As stated in [21], the vision is to assemble "application components into a network of services that can be loosely coupled to create flexible, dynamic business processes and agile applications that span organizations and computing platforms".

Services are "autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways" [21, p. 38]. When several services are combined to achieve a particular goal, it is called *Service Composition* [4, p. 55]. While there are several disciplines of Service

A. Madeira and M. A. Martins (Eds.): WADT 2022, LNCS 13710, pp. 121–140, 2023.
https://doi.org/10.1007/978-3-031-43345-0_6

Composition, we focus on Service Orchestration, i.e., creating new services "by combining several existing services in a process flow" [4, p. 57].

When composing or using existing services, we hopefully have multiple services fulfilling the functional requirements of our tasks. Beyond that, they typically stand out against each other in several non-functional attributes. While the price is an important aspect, they usually also differ in their *Quality of Service* (QoS) attributes, for example, latency or availability [16]. Therefore, an essential aspect of the Service Selection Problem [6, pt. II] is determining whether the QoS profile of a service satisfies the QoS requirements of a client.

Our approach is based on Constraint Programming (CP) [23] leaning on soft constraint solving to automate the process of selecting adequate services based on their QoS properties. Adding hard constraints to reduce the number of matching services is simple but might lead to either a still too extensive range of services or not a single one left if we overconstrain the problem. Soft constraints come in handy as the solver can omit them if the Constraint Satisfaction Problem (CSP) [11] would be overconstrained otherwise.

We present a tool, named QosAgg, for solving the service selection problem for composite services in a soft way. We leverage on MiniBrass, a tool presented in [26] that extends the MiniZinc [20] constraint modeling language and tool, providing various options to model and solve soft CSPs based on the unifying algebraic theory of Partial Valuation Structures (PVSs) [27]. Specifically, our approach provides the means for: 1) describing a service workflow over which the service selection has to be performed, 2) expressing QoS profiles associated with concrete services as values of its QoS attributes, 3) expressing QoS requirements as soft constraints over the aggregated value of QoS attributes along the execution of the workflow, 4) automatically finding the best (if any) assignment of services to tasks given the above set up.

In Sect. 2, we present our approach to the problem of selecting services to optimize global QoS requirements of a workflow. In Sect. 3 we introduce the MiniBrass modelling language. In Sect. 4 we show how to model and solve QoS aware service selection in MiniBrass. In Sect. 5 we perform preliminary performance experiments. Finally, in Sect. 6 we draw some conclusions and point out possible future lines of research.

Related Work. Our work consists of QoS-aware service selection for workflows, based on soft constraint solving. While optimization-based techniques can be separated into locally and globally optimizing ones, we focus on global optimization-based service selection, where the QoS of each service is considered pre-determined. In most cases, global optimization means that QoS has to be aggregated, Sakellariou and Yarmolenko [24] discuss how this can be done for several attributes.

There are knapsack and graph-path-finding-based approaches for modelling and solving the optimization problem [30]. Zheng, Luo, and Song [32] propose a colony-based selection algorithm applicable to multi-agent service composition [29]. We will delegate the solving of the problem to dedicated solvers but use a multidimensional, multiple-choice knapsack problem for modelling as well.

In most of the works that apply Constraint Programming for service selection, such as [14], only hard constraints are used. When soft constraints are used, the way to express preferences over solutions is quite limited. For example, [22] supports softness by assigning importance levels to constraints. Deng et al. [8] use constraint solving but concentrate on the domain of mobile cloud computing and therefore put emphasis on temporal constraints. Arbab et al. are working with (Soft) Constraint Automata [1,2,9] and use them for service discovery [3, 25]. However, Soft Constraint Automata turn out to be representable by soft constraint satisfaction problems (SCSPs) [2, sec. 6.1], and they concentrate on local optimization only.

Rosenberg et al. [22] provide an implementation as part of their VRESCo project [18] that also supports soft constraints [17], but only weighted ones as well. A more general formalization for soft constraints is c-semirings (Constraint Semirings) [5] that can also be used for service selection, as Zemni, Benbernou, and Carro [31] show, but without an implementation. We will fill this gap and provide flexible soft constraints for service selection in an easy-to-use manner for users with basic knowledge of constraint programming.

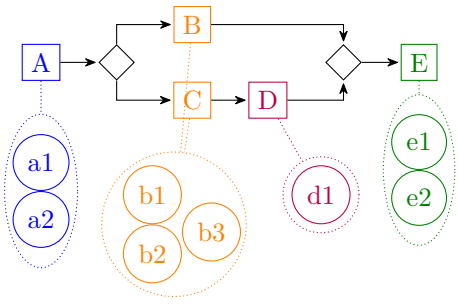
2 Service Selection for Composite Services

Composite services can often be described as workflows [4]. A workflow consists of one or multiple abstract services. An abstract service is a task that needs to be done. Each abstract service can be instantiated by any of a class of concrete ones that can fulfil the task. To avoid confusion, we refer to abstract services as *tasks* and to concrete services merely as *services*. The tasks in a workflow can be composed sequentially, in parallel, be subject to a choice and put within a loop.

The selection of services to fulfil the tasks in a workflow can be done in many ways. In our case, workflows are converted to execution plans defining the paths traversing the workflow. This allows the selection of adequate services for every task instance in the path, even admitting the selection of different services for performing different instances of the same task if it has to be executed more than once, e.g., in loops.

We'll start with a simple example workflow inspired by [19, Fig. 13.1].

Imagine a workflow with an initialization task A that can be done by two provisioning services a_1 and a_2 and a finalization task E with two eligible services e_1, e_2. In between, there are two possible paths: either task B is done, which has three provisions b_1 to b_3; or instead, task C with the same provisions is executed but then succeeded by task D that is done by the only available service d_1.



Definition 1. Workflow graphs *are defined by the following grammar:*

$$\begin{aligned}\langle \mathit{Task}\rangle &::= \mathit{Task\ name}\\ \langle G\rangle &::= \mathit{Null} \mid \langle \mathit{Task}\rangle \mid \langle G\rangle \to \langle G\rangle \mid \langle G\rangle|\langle G\rangle \mid \langle G\rangle + \langle G\rangle\\ &\quad\ \mathit{Additionally,}\ \langle G\rangle^n\ \mathit{serves\ as\ syntactic\ sugar}\ (n \in \mathbb{Z}_{>0}).\end{aligned}$$

where `A` $\to$ `B` *(or short:* `A B`*) denotes sequential composition,* `A | B` *parallel composition,* `A + B` *choice, and* $\texttt{A}^n$ *a loop with a fixed number of iterations* n*. The graph shown above looks like this:* `A` $\to$ `(B + C D)` $\to$ `E`*.*

As it is clear from the previous definition, we assume that iterations in a workflow graph are bounded, so every execution plan is finite and the procedure of service selection is safe from the pothole posed by the termination problem of unbounded iterations.

Definition 2 (Provisioning service description). *A service description consists of: 1) a service name, 2) a set of tasks it can be assigned to, and 3) a set of QoS attribute names associated with the specific values the service guarantees.*

For instance, recalling the previous example, provision `b3` can be assigned to tasks `B, C`, and have the following QoS attributes: `cost = 9`, `responsetime = 2`, `availability = 98`, `accuracy = 99.5`, etc.

Since we aim at performing a global selection we should be able to define preferences about the overall QoS of the composition. Therefore, we need a way to aggregate the QoS attributes of the individual services in a workflow configuration. For the QoS attributes that should be aggregated over the whole workflow, there needs to be information on how to aggregate them.

Definition 3 (Aggregation operator). *Each QoS attribute has two associated aggregation operators:*

- $\mathrm{agg}_{\to}$*, a binary operator for aggregating it over sequential composition, and*
- $\mathrm{agg}_{\|}$*, a binary operator for aggregating it over parallel composition.*

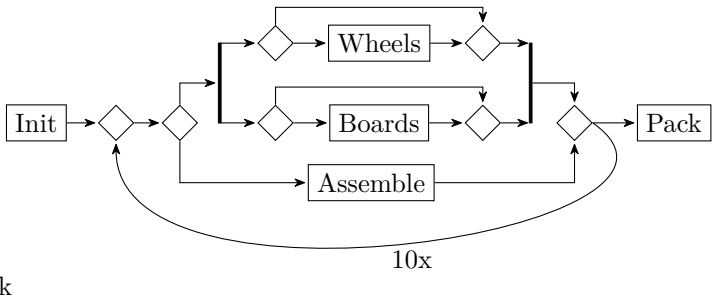
Next we introduce our running example.

Running Example: A company dedicated to manufacture skateboards rents two workstations in a co-working workshop.

Workflow. The company needs to rent storage for the wheels, boards, and the finished skateboards that it produces. The co-working workshop offers two rental models. In the first model, one can rent storage for precisely 10 or 15 items. In the second, one also has to decide in advance how many items to store but can rent storage for between 10 and 15 items. The second rental model is a bit more expensive on a per-item basis and takes a bit longer to set up.

Once the storage is rented, the company can start producing the boards. The work is organized in iterations. In each iteration, each workstation can work individually: one crafts wheels, the other one boards. Alternatively, work can be done together to assemble four wheels and one board to a skateboard. When assembling, one can decide to assemble three boards at once, which is a bit faster. Also, when crafting boards, you either can craft a single board or craft three boards of a different kind in a single iteration, which is a bit more time and cost-efficient, and the boards are a bit more pliable but also heavier. Regarding the wheels, we always create four wheels in a single iteration, but we can choose from four different kinds of wheels that differ in durability, friction, and cost.

When one workstation is done with the own task of an iteration, it waits for the other one to finish, too. After ten iterations, we pack the finished skateboards either not at all, using cardboard or in a wooden box. Cardboard—and wood even more—provides better protection but is more expensive and time-consuming.



Attributes. We care about the following global attributes that affect the overall outcome or the dependencies between tasks: `cost`, `time`, `storage`, `number of produced boards`, `number of produced wheels`, `number of finished products`.

3 Soft Constraint Solving with MiniBrass

MiniZinc [20] is a solver-independent constraint modeling language for describing CSPs and constraint optimization problems (COPs) and an associated tool which translates MiniZinc specifications into the lower-level solver input language FlatZinc, supported by numerous constraint solvers. MiniZinc is also used as a frontend for invoking the user-defined specific solver; which, in our case, will be Gurobi[1], a state-of-the-art commercial optimization solver. In contrast to traditional programming, where the programmer states what the program should do in order to compute the result, in constraint programming, the modeller only states what the solution must satisfy; then, a solver is responsible for coming up with potential solutions, checking them against the constraints in the model, and then returning any, or the best, solution.

MiniZinc differentiates between decision and parameter variables. While parameter variables are compile-time constant, i.e., their value is known even before the solver starts working, decision variables are the ones that the solver

[1] Available at https://www.gurobi.com.

can variate to come up with new solutions. MiniZinc supports a lot more capabilities, like arrays, quantifiers, or optimization, to name a few[2].

Example 1 demonstrates the usage of MiniZinc by showing a toy specification, together with its output and some considerations.

Example 1. MiniZinc specification:

```
1  set of int: DOM = 1..2; % DOM = {1, 2}
2  var DOM: x; var DOM: y; % x, y in DOM
3  constraint x!=y;
4  solve satisfy;
```

Line 1 defines a set `DOM` containing the integers 1 and 2, line 2 defines two decision variables `x` and `y` in `DOM`, line 3 constrains them to be different, and line 4 asks MiniZinc to solve the problem and return any satisfying solution.

MiniZinc's output after running:

```
x = 2;
y = 1;
```

Obviously, $(x, y) = (1, 2)$ would also have been a valid solution. Such a preference can be enforced by replacing the keyword `satisfy` by the objective function `minimize x` in the statement `solve` of line 4.

MiniBrass [26], also a modelling language equipped with an analysis tool, extends MiniZinc in two ways. On the one hand, it enriches the MiniZinc constraint modelling language with preference models containing soft constraints. Soft constraints are constraints that might be omitted if the problem would be unsatisfiable otherwise. MiniBrass supports a range of algebraic structures called Partial Valuation Structures (PVSes) [27] that enable the prioritization of constraints. On the other hand, MiniBrass implements a branch-and-bound search algorithm which iteratively generates MiniZinc models by adding constrains from the preference model whose solutions are considered subsequently better, according to the underlying PVS. In a sense, MiniBrass is providing the means for traversing the complete lattice of constraint systems, induced by the preference model [5, Thm. 2.9][3], and searching for an optimum solution. A more comprehensive explanation of the many algorithmic aspects involved in the implementation can be found in [26, p. 21].

While MiniBrass provides various predefined PVSes, e.g., for constraint preferences given as graph, fuzzy constraints, weighted CSPs, and many more, it also admits the definition of custom PVSes, if needed.

Definition 4 (Partial Valuation Structure – Definition 1, [27]). *A* partial valuation structure $M = (X, \cdot, \varepsilon, \leq)$ *is given by an* underlying set X*, an associative and commutative* multiplication *operation* $\cdot : X \times X \to X$*, a* neutral *element* $\varepsilon \in X$ *for* $\cdot$*, and a partial ordering* $\leq \subseteq X \times X$ *such that the multiplication* $\cdot$ *is*

[2] The interested reader might, however, have a look at the handbook https://www.minizinc.org/doc-latest/en/index.html.

[3] While [5, Thm. 2.9] is stated for c-semirings, PVSes can be converted to and created from c-semirings [5,13,26], another popular algebraic framework for soft constraints.

monotone in both arguments w.r.t. $\leq$*, i.e.,* $m_1 \cdot m_2 \leq m_1' \cdot m_2'$ *if* $m_1 \leq m_1'$ *and* $m_2 \leq m_2'$*, and* ε *is the top element w.r.t.* $\leq$*.*

We write $m_1 < m_2$ *if* $m_1 \leq m_2$ *and* $m_1 \neq m_2$*, and* $m_1 \parallel m_2$ *if neither* $m_1 \leq m_2$ *nor* $m_2 \leq m_1$*. We write* $|M|$ *for the underlying set and* $\cdot_M$*,* ε_M*, and* $\leq_M$ *for the other parts of* M*.*

Among the many PVSes already defined in MiniBrass we can find the PVS type `WeightedCsp` from [26, p. 27]. Such a PVS allows for assigning a weight to each of the soft constraints, which will act as preferences. In the resulting MiniZinc model, heavier constraints will be preferred over lighter ones.

Example 2 shows the use of PVSes for extending Example 1 by an instance of `WeightedCsp` in order to formalize a preference model.

Example 2. MiniBrass preference model:

```
1  include "defs.mbr";
2  PVS: prefer2 = new WeightedCsp("prefer2") {
3    soft-constraint xEquals2: 'x==2';
4    soft-constraint yEquals2: 'y==2' :: weights('2');
5  };
6  solve prefer2;
```

`Line 1` includes the standard MiniBrass definitions (`defs.mbr`) which, among others, allows the usage of `WeightedCsp`. The identifier `prefer2` in `line 2` is the name we choose for our PVS instance. `Lines 3` and `4` declare two soft constraints requiring `x` and `y` to be equal to 2 but establish `yEquals2` to be heavier (i.e., has weight 2 in contrast to 1 which is the default weight for the CSP). Therefore, the complete model consists of both, the hard constraints of the MiniZinc specification shown in Example 1 and the MiniBrass preference model shown above. As `x` and `y` have to be different according to the hard constraint, it is not possible to fulfill both soft constraint simultaneously. Even though $(x, y) = (2, 1)$ fulfils the hard constraints, the only admissible optimal solution is $(x, y) = (1, 2)$ because the soft constraint `yEquals2` is heavier than `xEquals2`.

New PVSes can be constructed by combining two PVSes using either the lexicographic or the Pareto product. The lexicographic combination `M lex N` prioritizes the ordering of solutions of `M` and only considers `N` when `M` cannot decide between two solutions. In the Pareto combination `M pareto N`, a solution is better than another if it is better for both `M` and `N`.

4 Modeling QoS-Aware Service Selection in MiniBrass

The tool we are presenting, named QosAgg, takes as inputs the workflow description including the quantitative attributes over which the QoS is to be evaluated, and the service definitions together with their possible assignments to tasks. Its output is the MiniZinc code containing the CSP to be solved including basic declarations, enums for tasks and services, decision variables assigning one service to every task and one branch per path choice. The model generated by QosAgg corresponds to a 0/1 multi-dimensional multi-choice knapsack problem [15,30]:

task instances are the bags, and we can put precisely one service into each bag. Next, one array per QoS attribute is created, containing the values for every service.

A key element in the translation to a CSP is, as we mentioned before, the aggregation of QoS attributes along the paths of the workflow in a way that makes possible to check the satisfaction of the desired constraints. From a theoretical point of view, bounded loops are no more than syntactic sugar, so we start by unfolding them in order to obtain the equivalent graph that can only be *null*, a *single task*, a *sequential composition*, a *parallel composition* or a *choice composition*. Then, for a graph G aggregation $q(G)$ is then defined recursively on its structure as follows:

- Let $\eta(T)$ denote the service chosen to perform the *single task* T,
- Let $\eta(G_0 + G_1 + \cdots + G_n)$ denote the specific subgraph selected by the choice,
- $q(null)$ yields the valuation which is $\text{agg}_{\rightarrow}()$ for all the QoS attributes,
- $q(T)$, with T a *single task*, yields the QoS contract of $\eta(T)$,
- $q(G_0 \rightarrow G_1 \rightarrow \cdots \rightarrow G_n)$ yields the valuation $\text{agg}_{\rightarrow}(q(G_0), q(G_1), \ldots, q(G_n))$,
- $q(G_0 \parallel G_1 \parallel \cdots \parallel G_n)$ yields the valuation $\text{agg}_{\parallel}(q(G_0), q(G_1), \ldots, q(G_n))$,
- $q(G_0 + G_1 + \cdots + G_n)$ yields the valuation $q(\eta(G_0 + G_1 + \cdots + G_n))$.

Essentially, q aggregates over the parallel and sequential composition using the corresponding aggregation operators. It deals with single tasks, and choices by using decision variables that let the solver make the best decision for the overall QoS. We continue by showing the modeling workflow of the running example introduced in Sect. 2.

Example 3 (A skateboard company). We start by showing in Listing 1.1 the input file for QosAgg containing the workflow definition, the provision contracts and the quantitative attributes that constitute the QoS model.

```
workflow wf {
  graph: Init -> ((Wheels? | Boards?) + Assemble)^10 -> Pack;

  provision bigStore for Init: cost = 60, time = 20, storage = 15;
  provision smallStore for Init: cost = 30, time = 10, storage = 10;

  provision badWheels for Wheels: cost = 5, time = 2, wheels = 4;
  provision okWheels for Wheels: cost = 5, time = 2, wheels = 4;
  provision expensiveWheels for Wheels: cost = 10, time = 2, wheels = 4;
  provision goodWheels for Wheels: cost = 5, time = 2, wheels = 4;

  provision singleBoard for Boards: cost = 7, time = 3, boards = 1;
  provision threeBoard for Boards: cost = 16, time = 10, boards = 3;

  provision singleAssembly for Assemble: cost = 2, time = 4, products = 1,
      wheels = -4, boards = -1;
  provision threeAssembly for Assemble: cost = 6, time = 10, products = 3,
      wheels = -12, boards = -3;

  provision noPacking for Pack: cost = 0, time = 0;
  provision woodPacking for Pack: cost = 20, time = 10;
  provision cardboardPacking for Pack: cost = 3, time = 3;

  attribute cost of var int; aggregation cost: sum;
  attribute time of var int; aggregation time: sum, max;
```

```
  attribute boards of int default 0; aggregation boards: sum;
  attribute wheels of int default 0; aggregation wheels: sum;
  attribute products of int default 0; aggregation products: sum;
  attribute storage of var int default 0;
};
```

Listing 1.1. QoS model

If we run MiniZinc to solve the CSP produced by QosAgg, it will output a statement displaying a solution to the problem including a path across the workflow together with the selected services for each task instance in the path, and the aggregated value for each QoS attribute for that selection.

Arbitrary hard constraints can be added on top of the basic CSP problem output by QosAgg in order to force MiniZinc to find more specific solutions satisfying both, the basic model, and the newly added hard constrains. For example, we can enrich our model by defining the notion of `profit` by means of fixing the retail price (in this case at 25) and considering the aggregated cost and the aggregated number of finished products along the selected path. This will make MiniZinc compute the value of the variable `profit` enabling, for example, the possibility of enforcing a lower bound for its value stating that we only accept solutions leading to a profit greater than such a bound (shown in Listing 1.2). This is done by feeding MiniZinc with both, the basic MiniZinc model obtained from QosAgg with the following handcrafted MiniZinc specification:

```
int: price = 25;
int: bound = 10;
var int: profit = price * wf_aggregated_products - wf_aggregated_cost;

constraint profit > bound;
```

Listing 1.2. MiniZinc constrain model

Analysing the resulting model will lead to any solution (i.e., a path in the workflow and an assignment of services to tasks) in which the value calculated for `profit` is greater than 10. MiniZinc can also be run with the statement `solve maximize profit;` forcing the tool to find an optimum solution in which the value of `profit` is not only greater than 10, but also the maximum possible.

Going further, we propose to aim at a richer form of constraints. Adding soft constraints to our model allows to, for example, force the solvers to search for solutions that increase profit and decrease time consumption. This can be done by writing a MiniBrass preference model resorting to two instances of the predefined PVS type `CostFunctionNetwork` and the lexicographical product for combining them as shown in Listing 1.3.

```
PVS: profit = new CostFunctionNetwork("profit") {
  soft-constraint profit: '500-profit';
};
PVS: time = new CostFunctionNetwork("time") {
  soft-constraint time: 'wf_aggregated_time';
};
solve profit lex time;
```

Listing 1.3. MiniBrass preference model

The process continues by feeding MiniBrass input the preference model shown above, and the basic MiniZinc resulting from combining: 1) the basic model output by QosAgg from the original model, enriched with 2) the additional handcrafted hard constraints of a choice.

It will then initiate the search for an optimum solution to the Soft CSP. As we mentioned before, this is done by applying a branch-and-bound searching algorithm over the complete lattice of constraint systems, induced by the PVS formalizing the preference model. The procedure implemented in MiniBrass will iteratively generate MiniZinc CSPs by adding constraints forcing any solution to be better than the one found in the previous iteration. In each iteration MiniZinc is run finding such solution. The iterative process is performed until the CSP gets unsatisfiable, at which point, an optimal solution has been found in the previous iteration.

Running MiniBrass on: a) the combination of the output of running QosAgg on the model shown in Listing 1.1 and the MiniZinc constrain model shown in Listing 1.2, and b) the MiniBrass preference model shown in Listing 1.3, yields the statement shown in Listing 1.4.

```
Profit: 13
Selection graph for wf:
    Init=bigStore → (Wheels=goodWheels | ) → (Wheels=goodWheels | ) → (
        Wheels=goodWheels | Boards=threeBoard) → Assemble=threeAssembly → (
        Wheels=goodWheels | Boards=threeBoard) → (Wheels=goodWheels | ) →
        Assemble=singleAssembly → (Wheels=goodWheels | ) → Assemble=
        singleAssembly → Assemble=singleAssembly → Pack=cardboardPacking
Aggregations for wf:
    cost: 137
    time: 73
    boards: 0
    wheels: 0
    products: 6
```

Listing 1.4. MiniZinc solution with aggregation values

The solution has a profit value of 13, workflow is displayed with the selected services for each task instance, and the aggregated value obtained for each QoS attribute is shown. The total cost of the solution is 137, the total time is 73, and the total number of skateboards produced is 6. The attributes `boards` and `wheels` are used to keep track of the number of boards and wheels produced. When the task `Assemble` is executed to produce skateboards it consumes `boards` and `wheels` and produces `products`. A final number of 0 for `boards` and `wheels` means that all the boards and wheels produced have been consumed to produce skateboards.

4.1 Adding Checkpoints to QosAgg Workflows

Up to this point, we showed how to model the problem of assigning services to tasks organized in a complex workflow, and how it can be solved based on the satisfaction of a combination of: 1) hard constraints added to the basic model, the latter obtained from the description of the workflow, the declaration of the QoS attributes and the declaration of services capable of performing each of the

tasks, and 2) soft constrains declared as a preference model through the use of PVSes.

This approach yields a framework in which it is possible to reason about the overall aggregated-by-attribute QoS of workflows and the local QoS of the distinct tasks, but we lack everything in between. This void might lead to a problem when a desired property is supposed to hold after the execution of a specific part of a workflow which is not after its completion. Consider the example of attributes that do not exclusively grow (resp. shrink), but that can both grow and shrink, and we need to preserve certain invariants regarding greater and lower bounds for such attributes. A classic example is that of producers and consumers of resources.

Example 4. Imagine a workflow graph $A \to B \to C$ where tasks A and C are meant to produce some resource, and B consumes it. Let there be services a_1, a_2 for task A, b_1, b_2 for B, and c for C, with QoS attributes "cost" and "resource" (interpreted as the cost associated to the execution of the service, and the resources produced/consumed by the service) with addition as aggregation function, and the following QoS contracts:

	a_1	a_2	b_1	b_2	c
cost	1	2	1	2	1
resource	1	2	−2	−1	2

Then, if we solve optimizing aiming at the lower overall cost, we end up with the selection a_1, b_1, and c with aggregated cost 3. It is clear that this solution is not satisfying as service a_1 only produces one resource item, but b_1 consumes two. Adding a constraint to the overall aggregation of the resource attribute is not of any use because service c adds two more resource items at the end, compensating the (infeasible) "debt" caused by b.

Example 4 exposes the need of some form of constraints over the aggregated value of QoS attributes at chosen points within the workflow. Such points in the execution of a workflow are referred to as "checkpoints" and are placed directly before and after tasks. They allow us to specify *invariants* by addressing all the relevant checkpoints in a certain fragment of interest of the workflow, or to specify pre-/post-conditions for specific tasks only by addressing the checkpoints appearing before and after such a task. Figure 1 illustrates this.

Aggregation on Checkpoint. Checkpoints mark those points in the workflow where constraints are plausible to be placed. Adding constraints at checkpoints requires the capability of aggregating the values of QoS attributes up to the specific checkpoint of interest. The reader should note that the definition of the aggregation presents no further difficulty with respect to what we discussed at the beginning of the present section but with the sole difference that now the

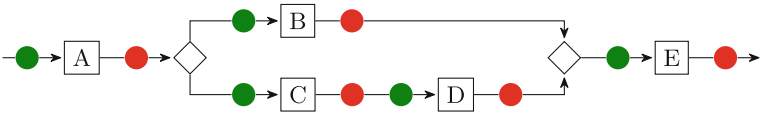


Fig. 1. Checkpoints in a workflow. ● are pre-conditions, ● post-conditions. (Color figure online)

evaluation is only performed over the maximal subgraph starting at the beginning of the workflow, and leading to the checkpoint one is interested in as an ending point.

Constraints on Checkpoints. Checkpoints allow us a smoother implementation of various constraints. Going back to our running example, we can observe that there is an actual risk of: 1) the sum of the produced wheels, boards, and finished skateboards in the storage might exceed the capacity we booked, or 2) the numbers of wheels, boards, and skateboards might be negative;

or, at least, there is no formal impediment for any of those situations to occur. Therefore, we would like to guarantee that none of those situations happens to be true at any point in the path selected as a solution. The following constraint shows how checkpoints help in enforcing this type of properties:

```
constraint forall(cp in wf_all_checkpoints)(
  wf_checkpoints_boards[cp] + wf_checkpoints_wheels[cp] +
      wf_checkpoints_products[cp] <= storage /\
  wf_checkpoints_boards[cp] >= 0 /\ wf_checkpoints_wheels[cp] >= 0 /\
      wf_checkpoints_products[cp] >= 0
);
```

In the previous constraint `wf_all_checkpoints` is the designated name for the set containing all the checkpoints of the workflow, and `wf_checkpoints_boards`, `wf_checkpoints_wheels` and `wf_checkpoints_products` are arrays containing the aggregated attribute value up to every checkpoint in `wf_all_checkpoints`.

Finally, by resorting to this type of constraints we can recall Example 4 and provide an elegant solution for the problem we used as motivation. The following constraint is what we need: "`constraint forall(cp in wf_all_checkpoints)(wf_checkpoints_resource[cp] >= 0);`".

Loops introduce a complex control flow structure that requires special treatment in order to provide a flexible way of establishing constraints allowing them to restrict all the iterations or just a single one, as shown in the following example. Let a workflow have graph $(A^3 \parallel B)^2$ and a single QoS attribute named `resource`. As tasks in a path are named according to their concrete instance once the iterations are unfolded, all of them have their own associated checkpoints so we can, for example, ensure that we start with at least five resource items in the first iteration by adding the following constrain: "`constraint wf_checkpoints_resource[wf_A_pre_1_1] >= 5;`".

Analogously, "`wf_A_pre_2_3`" would be the name for the checkpoint for the last iteration. A constraint ensuring that after executing (any instance of) *B* there are less than five resource items can be stated as follows: "`constraint forall(cp in wf_checkpoints_B_post)(wf_checkpoints_resource[cp] < 5);`".

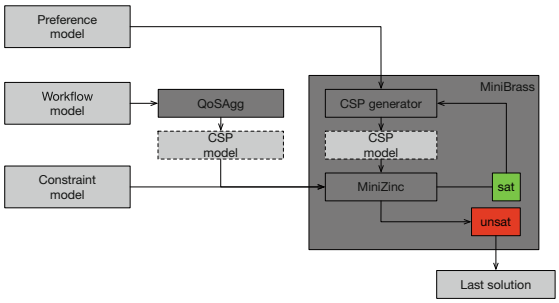
The case of workflows containing choices present a different, and very important issue. Consider workflow "*Give* + *Take*" and again a single QoS attribute named `resource`. The services for *Give* all produce items; the services for *Take* all consume them. Again we want to ensure that no resource is used before it has been produced. Adding the constraint "`constraint forall(cp in wf_all_checkpoints)(wf_checkpoints_resource[cp] >= 0)`" solves the problem but only partially. Note that, as there is no loop, the only reasonable choice is the path executing *Give* and omitting task *Take*, and that is the right solution. However, MiniZinc yields that the problem is unsatisfiable; this is because `wf_all_checkpoints` also contains the checkpoint `wf_Take_post`, and there the resource balance is negative. Nevertheless, when choosing the path with *Give*, we can ignore that checkpoint as the execution never even comes across task *Take*.

This is a problem regarding the reachability of specific points. To solve this issue we added expressions for each task instance stating whether it is reachable, i.e., part of the selected path, or not. We use these expressions to include only those checkpoints in the predefined checkpoint sets that are part of the selected path. For a task instance to be reachable, all the choices that it is part of need to select the branches leading towards the instance.

Once again, for the code generation, we recursively descend in the workflow graph. Each time we come across a choice composition, we remember the name of its choice decision variable and the branch we descended into. When we reach a single task, the conjunction of all choice variables we came across having the value required for the branch we went into gives us the reachability expression. In the case of the task Take in motivating situation described above, this would be: "`choice1 == 2`". Therefore, the checkpoint set `wf_all_checkpoints` is generated by filtering all the checkpoints for reachability. However, individual checkpoints, like "`wf_Take_post`" in our example, require manual handling. For example, the "`constraint wf_checkpoints_resource[wf_Take_post] >= 0`" has to hold even if "`wf_Take_inst`" is not reachable. One way to solve this is to only "enable" constraints when the instance is reachable. This is done by resorting to the assertion "`wf_reachable`" with which it is possible to state the constraint: "`constraint wf_reachable[wf_Take_inst] -> wf_checkpoints_resource[wf_Take_post] >= 0;`".

4.2 Toolchain Architecture

In the figure, we depict the architecture of the toolchain we propose for solving the problem of QoS-aware service selection for tasks organized as complex workflows described at the beginning of this section.



Dark grey nodes symbolize tools and light grey ones are files; among the latter, those with solid outline are either the model, or the output statement, and those with the dashed outline are intermediate files resulting from processing the model. The model consists of: 1) the *workflow model* containing: a) the graph of tasks, b) the QoS attributes, each of them with their corresponding aggregation functions for both, parallel composition and sequential composition, and c) the services' QoS specification and possible assignment to tasks; 2) the *constraint model* consisting of the hard constraints the user wants the solution to satisfy, and 3) the *preference model* consisting of the soft constraints the user wants to guide the search for a solution.

The tools include: 1) QosAgg that takes the workflow model as input and produces a file containing the basic CSP model containing the specification of the corresponding 0/1 multi-dimensional multi-choice knapsack problem, 2) MiniBrass that takes the CSP model resulting from combining the output of QosAgg and the constraint model, and the preference model, and implements the branch-and-bound search algorithm for incrementally finding the best solution, according to the preference model, and 3) MiniZinc that runs the solver over the complete model in order to find the optimum solution.

5 Preliminary Performance Analysis

In this paper we proposed a toolchain for QoS-aware service selection for tasks organized as complex workflows. Among the different tools involved in it, we were responsible only for the development of QosAgg. On the one hand, an exclusive performance analysis of QosAgg does not lead to any significant conclusion because, as we mentioned before, it is a simple parsing process translating workflow models to Soft CSP; on the other hand, any discussion on the theoretical complexity/empirical study of the toolchain formed by MiniBrass, MiniZinc and Gurobi on arbitrary Soft CSP[4], does not provide the right insight on the actual performance of such tools in analysing the Soft CSPs obtained from QosAgg. For

[4] The interested reader is pointed to [5,28] for the results associated to the theoretical complexity of the formal framework underlying MiniBrass and to [26, section 5] for an empirical evaluation. In the case of the complexity associated to the use of MiniZinc there is not much to be said about the translation to FlatZinc (i.e., its target language) because most of the computational effort resides in the execution of the solver [20]. Regarding Gurobi; a comprehensive empirical study against the SAS solvers, available at https://www.sas.com, running over the Mittelmann's benchmark can be found in [12].

this reason, we chose to perform an empirical performance study of the complete toolchain we proposed as a blackbox.

For comparability reasons, the workflow model, the constraint model and preference models are synthetically generated in a specific way to be explained below. All the experiments are carried out using MiniZinc 2.6.4 with the proprietary solver Gurobi 9.5.2 on a machine having an Apple M1 chip with eight cores and 16 GB RAM on a 64 bit macOS Monterey.

This experimental study pretends to shed some light on how the structure of the workflow drives the complexity of the analysis so we devised experiments aiming at revealing the compositional nature of the computational effort required to solve a problem. To this end we: 1) performed an empirical study of the cost associated to solving Soft CSPs obtained from workflows consisting of single tasks whose complexity varies according to: a) the number of service providers, and b) the number of quantitative attributes involved in the model, 2) studied the correlation between the cost associated to the analysis of the composition of workflows (sequential, parallel and choice) and a function of the costs associated to the analysis of the workflows involved in such a composition. In this case we varied the amount of workflows (only considering simple tasks) in the composition.

The property under analysis in all cases is the lex composition of the maximization of the value of each attribute. We start by identifying the impact of the number of attributes and providers on the computational cost of solving the optimum service assignment for workflows consisting of a single task. To this end we fixed the structure of the workflow, the hard constrains and the soft constrains in order to obtain a family of Soft CSPs whose analysis can reflect the growth in the computational effort required while a problem gets bigger, either in terms of the amount of attributes or the amount of service providers. In order to ameliorate statistical deviations, we ran the tool over 10 randomly generated instances of workflows consisting of a single task and varying the number of attributes ranging from 10 to 100 stepping by 10 and providers ranging from 1 to 2000 stepping by 100, and reported the average of the values obtained in the runs. From the experimental data we can derive the following observations: 1) the computational cost associated to QosAgg, when varying the amount of service providers, grows linearly in all the cases with[5] $R^2 \geq 0.99$, 2) the computational cost associated to MiniBrass, when varying the amount of service providers, grows polinomially (with grade 2) with $R^2 \geq 0.79$, with the exceptions of the experiments for 1 attribute, in which $R^2 = 0.7462$; the average R^2 is 0.8901, 3) the computational cost associated to QosAgg, when varying the amount of attributes, grows linearly in all the cases with $R^2 \geq 0.9$, 4) the computational cost associated to MiniBrass, when varying the amount of attributes, grows polynomially (with grade 2) with $R^2 \geq 0.74$; the average R^2 is 0.857, 5) the computational cost associated to QosAgg is at most around 30% of the total cost of analysis.

[5] R squared, denoted R^2, is the *coefficient of determination* that provides a measure of how well the model fits the data.

We continue by analyzing the computational cost associated to the workflow composition operators (i.e., sequential, parallel and choice composition). We generated 10 sets containing 10 workflows consisting of a single task, 100 providers and 50 QoS attributes. In order to understand how the size of the composition impacts the cost of analysis, each set is used to conduct an experiment in which we subsequently increment the size of the composition from 1 to 10 subworkflows. In both parallel and sequential composition we used `max` as the aggregation function. From the previous experimental data we can derive the following observations about the behaviour of the sequential and parallel composition: 1) the computational cost associated to the execution of QosAgg, when varying the amount of workflows in the composition, grows linearly in average and in all the individual cases. In the average case the fitting has $R^2 \geq 0.99$, 2) the computational cost associated to the execution of MiniBrass, when varying the amount of workflows in the composition, grows exponentially both in average and in all the individual cases. In the average case the fitting has $R^2 \geq 0.98$, and 3) the computational cost associated to the execution of MiniBrass excedes the timeout of one hour for cases of compositions consisting of 8 or more workflows (except for 3 and 2 cases for sequential and parallel composition respectively).

The results for sequential and parallel composition are similar, this is due to the fact that in both cases we are using the same aggregation function, which yields the same minizinc model. The reader should also note that the analysis time may vary a lot depending on many other factors; we can identify some obvious ones like: 1) the choice, and diversity, of aggregation functions associated to the quantitative attributes, 2) the hard and soft constraints, which can severely influence the behaviour of the analysis tools, and 3) how intricate is the structure of the workflow,

among others. In the case of the choice composition operator we can derive the following observations: 1) the computational cost associated to the execution of QosAgg, when varying the amount of workflows in the choice composition, grows linearly in average and in all the individual cases. In the average case the fitting has $R^2 \geq 0.99$, and 2) the computational cost associated to the execution of MiniBrass, when varying the amount of workflows in the choice composition, grows polinomially (with grade 2) both in average and in all the individual cases. In the average case the fitting has $R^2 \geq 0.99$.

In summary, the execution cost of QosAgg increases linearly and accounts for a relatively small portion of the overall analysis cost. On the other hand, the execution cost of MiniBrass exhibits exponential growth in the case of parallel and sequential composition, while demonstrating polynomial growth in the case of choice composition. Unsurprisingly, the cost of executing MiniBrass constitutes the majority of the total analysis cost.

6 Conclusions and Further Research

We presented a toolchain supporting optimum QoS-aware service selection for tasks organized as workflows, based on soft constrain solving. QosAgg is used to

generate a skeleton MiniZinc model from workflow specifications (i.e., a description of the workflow, an enumeration of the QoS attributes together with their corresponding aggregation operator, and the list of providers for each task, including their QoS profile, expressed as values for the QoS attributes). Such a MiniZinc model contains, non-exclusively, decision variables corresponding to aggregations of the QoS attributes that can be used to enforce additional constrains over specific points of the workflow. On top of the resulting MiniZinc CSP, it is possible to add soft constrains resulting in a Soft CSP that can be solved using MiniBrass. We performed a preliminary performance analysis under the hypothesis that the computational cost of solving the Soft CSPs generated is driven, and compositionally determined, by the composition operators used to create workflows. Such study exhibited the impact of the exponential nature of solving the Soft CSPs by MiniBrass on the overall performance of the toolchain.

QosAgg creates decision variables for all possible path and service selections. These might be too many for MiniZinc to handle for more extensive use cases; in that case, it might be necessary to make MiniZinc evaluate only one specific path choice at a time and repeat that for all the possible paths in an iterative process in order to obtain scalability. Moreover, we focused on offline optimization only (i.e., all information had to be provided from the beginning). In reality, one might only have estimations of the values as QoS contracts whose real run-time value might affect future decisions leading to a dynamic notion of optimum relative to the online behavior of the selected providers. There is on going research about how to integrate offline and online decision-making [7].

Finally, there are many situations our workflows cannot model directly and need to be sorted out manually that are left for further research. To name a few: there are no built-in conditional path choices that depend on aggregated values. Support for compensation actions [10] would also be helpful, e.g., for the case where services can fail. Services at the moment are assumed to have constant QoS attributes across all executions. Support for probabilistic decisions would make it much easier to model decisions that we cannot influence, e.g., because the user of the composite service makes them, etc.

References

1. Arbab, F., Baier, C., Rutten, J., Sirjani, M.: Modeling component connectors in reo by constraint automata: (extended abstract). Electron. Notes Theor. Comput. Sci. **97**, 25–46 (2004). https://doi.org/10.1016/j.entcs.2004.04.028
2. Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: ter Beek, M.H., Lohmann, N. (eds.) WS-FM 2012. LNCS, vol. 7843, pp. 118–133. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38230-7_8. ISBN 978-3-642-38230-7
3. Arbab, F., Santini, F., Bistarelli, S., Pirolandi, D.: Towards a similarity-based web service discovery through soft constraint satisfaction problems. In: Proceedings of the 2nd International Workshop on Semantic Search over the Web, ICPS Proceedings, New York, NY, USA. Association for Computing Machinery (2012). https://doi.org/10.1145/2494068.2494070. ISBN 978-1-4503-2301-7

4. Baryannis, G.: Service composition. In: Papazoglou, M.P., Pohl, K., Parkin, M., Metzger, A. (eds.) Service Research Challenges and Solutions for the Future Internet. LNCS, vol. 6500, pp. 55–84. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17599-2_3. ISBN 978-3-642-17599-2
5. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM **44**(2), 201–236 (1997). https://doi.org/10.1145/256303.256306. ISSN 0004–5411
6. Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.): Web Services Foundations. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7. ISBN 978-1-4614-7517-0
7. De Filippo, A., Lombardi, M., Milano, M.: Integrated offline and online decision making under uncertainty. J. Artif. Int. Res. **70**, 77–117 (2021). https://doi.org/10.1613/jair.1.12333. ISSN 1076–9757
8. Deng, S., Huang, L., Wu, H., Wu, Z.: Constraints-driven service composition in mobile cloud computing. In: 2016 IEEE International Conference on Web Services (ICWS), pp. 228–235 (2016). https://doi.org/10.1109/ICWS.2016.37
9. Dokter, K., Gadducci, F., Santini, F.: Soft constraint automata with memory. In: de Boer, F., Bonsangue, M., Rutten, J. (eds.) It's All About Coordination. LNCS, vol. 10865, pp. 70–85. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90089-6_6. ISBN 978-3-319-90089-6
10. El Hadad, J., Manouvrier, M., Rukoz, M.: Tqos: transactional and qos-aware selection algorithm for automatic web service composition. IEEE Trans. Serv. Comput. **3**(1), 73–85 (2010). https://doi.org/10.1109/TSC.2010.5
11. Freuder, E.C., Mackworth, A.K.: Constraint satisfaction: an emerging paradigm. In: Handbook of Constraint Programming, vol. 2, 1 edn. (2006). ISBN 978-008-04-6380-3
12. Helm, W.E., Justkowiak, J.-E.: Extension of Mittelmann's benchmarks: comparing the solvers of SAS and Gurobi. In: Fink, A., Fügenschuh, A., Geiger, M.J. (eds.) Operations Research Proceedings 2016. ORP, pp. 607–613. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-55702-1_80
13. Hosobe, H.: Constraint hierarchies as semiring-based csps. In: 2009 21st IEEE International Conference on Tools with Artificial Intelligence, pp. 176–183. IEEE (2009). https://doi.org/10.1109/ICTAI.2009.43
14. Lecue, F., Mehandjiev, N.: Towards scalability of quality driven semantic web service composition. In: 2009 IEEE International Conference on Web Services, pp. 469–476. IEEE (2009). https://doi.org/10.1109/ICWS.2009.88. ISBN 978-0-7695-3709-2
15. Martello, S., Toth, P.: Algorithms for knapsack problems. In: Martello, S., Laporte, G., Minoux, M., Ribeiro, C. (eds.) Surveys in Combinatorial Optimization, number 132 in North-Holland Mathematics Studies, North-Holland, pp. 213–257 (1987). https://doi.org/10.1016/S0304-0208(08)73237-7
16. Menascé, D.A.: Qos issues in web services. IEEE Internet Comput. **6**(6), 72–75 (2002). https://doi.org/10.1109/MIC.2002.1067740. ISSN 1941–0131
17. Meseguer, P., Rossi, F., Schiex, T.: Soft constraints. In: Handbook of Constraint Programming, vol. 9, 1 edn., pp. 281–328 (2006). ISBN 978-008-04-6380-3
18. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-end support for qos-aware service selection, invocation and mediation in vresco. Technical report, Vienna University of Technology (2009). https://dsg.tuwien.ac.at/Staff/sd/papers/TUV-1841-2009-03.pdf

19. Moghaddam, M., Davis, J.G.: Service selection in web service composition: a comparative review of existing approaches. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) Web Services Foundations, pp. 321–346. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_13
20. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38. ISBN 978-3-540-74970-7
21. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: state of the art and research challenges. Computer **40**(11), 38–45 (2007). https://doi.org/10.1109/MC.2007.400. ISSN 1558–0814
22. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An end-to-end approach for qos-aware service composition. In: 2009 IEEE International Enterprise Distributed Object Computing Conference, pp. 151–160. IEEE (2009). https://doi.org/10.1109/EDOC.2009.14. ISBN 978-0-7695-3785-6
23. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, 1 edn. Elsevier Science Inc., Amsterdam (2006). ISBN 978-008-04-6380-3
24. Sakellariou, R., Yarmolenko, V.: On the flexibility of ws-agreement for job submission. In: Proceedings of the 3rd International Workshop on Middleware for Grid Computing, ICPS Proceedings. Association for Computing Machinery (2005). https://doi.org/10.1145/1101499.1101511. ISBN 978-1-59593-269-3
25. Sargolzaei, M., Santini, F., Arbab, F., Afsarmanesh, H.: A tool for behaviour-based discovery of approximately matching web services. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 152–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_11. ISBN 978-3-642-40561-7
26. Schiendorfer, A., Knapp, A., Anders, G., Reif, W.: MiniBrass: soft constraints for MiniZinc. Constraints **23**(4), 403–450 (2018). https://doi.org/10.1007/s10601-018-9289-2
27. Schiendorfer, A., Knapp, A., Steghöfer, J.-P., Anders, G., Siefert, F., Reif, W.: Partial valuation structures for qualitative soft constraints. In: De Nicola, R., Hennicker, R. (eds.) Software, Services, and Systems. LNCS, vol. 8950, pp. 115–133. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15545-6_10. ISBN 978-3-319-15545-6
28. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: hard and easy problems. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 1995, Montréal, Québec, Canada, 20–25 August 1995, vol. 2, pp. 631–639. Morgan Kaufmann (1995)
29. Wei, L., Junzhou, L., Bo, L., Xiao, Z., Jiuxin, C.: Multi-agent based QoS-aware service composition. In: 2010 IEEE International Conference on Systems, Man and Cybernetics, pp. 3125–3132. IEEE (2010). https://doi.org/10.1109/ICSMC.2010.5641725
30. Tao, Yu., Zhang, Y., Lin, K.-J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans. Web **1**(1), 6-es (2007). https://doi.org/10.1145/1232722.1232728. ISSN 1559–1131
31. Zemni, M.A., Benbernou, S., Carro, M.: A soft constraint-based approach to QoS-aware service selection. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 596–602. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17358-5_44. ISBN 978-3-642-17358-5

32. Zheng, X., Luo, J.Z., Song, A.B.: Ant colony system based algorithm for qos-aware web service selection. In: Kowalczyk, R. (ed.) Grid Service Engineering and Management "The 4th International Conference on Grid Service Engineering and Management" GSEM 2007, number 117 in Lecture Notes in Informatics, Bonn, Germany, pp. 39–50. Gesellschaft für Informatik e. V. (2007). https://dl.gi.de/server/api/core/bitstreams/4cefa9ab-94e1-4d82-b2ea-4d8ea1041838/content. ISBN 978-3-88579-211-6