INFORMATICA, 2010, Vol. 21, No. 4, 487–504

QOS-Aware Composition of Enterprise System's Components: Constraint Logic Programming Approach

Jeremy BESSON, Albertas ČAPLINSKAS
Vilnius University Institute of Mathematics and Informatics
Akademijos 4, LT-08663 Vilnius, Lithuania
e-mail: contact.jeremy.besson@gmail.com, albertas.caplinskas@mii.vu.lt

Received: May 2010; accepted: October 2010

Abstract. Enterprise systems should be assembled out of components and services according to an orchestration schema and taking into account not only functional requirements but also the resulting Quality of Service (QoS). In other words, QoS-aware composition of services and components must be performed. The problem is to find which components or services have to be employed that the resulting system would optimize some QoS attributes while satisfying some other QoS constraints. The paper proposes to use the Constraint Logic Programming approach to solve this problem, that is, we see this problem as a discrete optimization and satisfaction problem.

Keywords: enterprise systems, service-oriented engineering, quality of services, composition of services, discrete optimization, constraint logic programming.

1. Introduction

At present, more and more often advanced enterprise systems are composed of distributed commercial components and services delivered via Internet (Götze *et al.*, 2009). In the traditional branches of engineering such as manufacturing, assembling of complete systems out of prefabricated parts is of primary interest for creating easier-made and economical products. This is possible because of the existence of standardized components that meet predefined functional and non-functional requirements. However, at present we cannot yet enjoy the same luxury in software development. This is mainly because non-functional properties of components and services are often specified not exhaustive enough and because of difficulties to predict the quality of software that is composed even of parts that have a well-defined quality of delivered services. If the specification of Quality of Service (QoS) is not exhaustive or can be interpreted differently, it becomes impossible to compare the QoS, offered by a set of different components with equivalent functionalities and to predict the quality of a software system obtained by composing multiple components and services. However, the ability to predict automatically the QoS of the whole system is critical, if we think of the potential of the component-based and service-oriented software engineering paradigms, especially, in the environment of the Internet of Services.

According to ISO 9126 (ISO/IEC, 2001), an international standard for the evaluation of software, the QoS is "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs". The expected benefits of QoS specifications are for both users and providers. On the one hand, users want to select the best provider with regard to their needs. On the other hand, providers want to better advertise their services or components. QoS serves as a tool to differentiate providers that are providing similar functionalities. However, it is not realistic to expect specifications to be complete with respect to all such possible non-functional properties, due to the great variety of services and, consequently their quality characteristics. The efforts have been concentrated on identifying some common QoS attributes such as cost, availability, reliability, response time, latency, performance, security, accessibility, robustness, flexibility, and accuracy.

According to Plebani (2006), QoS is a combination of three types of qualities: (1) quality in use related to the quality perceived by the user, (2) internal quality regardless of the context in which it is used, and (3) external quality related to the context in which it is used. Numerous languages, policies and models have been proposed to specify, design and manage the QoS. Some examples are: credentials, a contract-based approach for specification of QoS (Shaw, 1996), a general purpose QoS modelling language QML (Froulund and Koistinen, 1998), an OWL ontology to describe the Web service OWL-S (W3C, 2004), UniFrame, a generative domain-specific model, which is used to generate the glue/wrapper interfaces between the required components (Raje, 2000), UML profile for QoS and Fault Tolerance (OMG, 2000), service component architecture SCA Policy (BEA Systems *et al*., 2007a, 2007b), a framework for providing QoS in network-centric distributed applications, including the embedded ones QuO and the QDL language for contract-based specifications with grammars (Pal *et al*., 2000; Ludwig *et al*., 2003; Lamanna *et al*., 2003), XML-based languages to describe service level agreements.

Contracts lay the foundations for negotiating and ensuring QoS. They support the relationship between a provider and a consumer. Service Level Agreement (SLA) is a special type of contract for QoS. TeleManagement Forum defines SLA as "a formal negotiated agreement between two parties, sometimes called a service level guarantee. It is a contract (or part of it) that exists between the service provider and the customer, designed to create a common understanding about services, priorities, responsibilities" (TeleManagement Forum, 2004). There are many different formats, but mainly SLA includes parties involved in the process, the service description (usually in an informal way as a text), functionality of service, SLA parameters (defining QoS attributes), service level objectives and QoS guarantees. Although today SLAs are negotiated often by face to face interaction, the main challenge is to prepare and sign SLAs automatically. A SLA should be negotiated also in case a composite of services should be delivered.

Services and components are usually composed at run-time. Several mechanisms exist to compose services, such as orchestration (Kühne *et al*., 2005), choreography (W3C, 2005), and pipe and filter which can direct the output of one processing element into the input of another processing element. BPEL4WS (OASIS, 2007; Business Process Execution Language for Web Services) and WSCI (W3C, 2002; Web Service Conversation

Interface) are examples of Web service orchestration languages. WS-CDL (Web Service Choreography Description Language; W3C, 2005) is an example of the choreography language. According to Mennie and Pagurek (2000), composite services at run-time can be produced using different techniques.

> *"In the first approach, two or more components collaborate while each component remains distinct, and potentially distributed, within a network. To facilitate this, a new common interface must be constructed at runtime which allows other services to interact with this set of collaborating service components as if it was a single service. The construction of this interface can be realized with the support of a service composition architecture. In the second approach, a new composite service is formed where all of the functionality of that service is contained in a single new component. This new service must be a valid service, capable of the basic set of operations that all other services can carry out."*

In the service oriented architecture world, services can be composed using late-binding mechanisms and predefined orchestration techniques. Appropriate services can be discovered using the standards as WSDL, UDDI and SOAP (W3C, 2004).

At a high level of abstraction, when considering only the flow of data between components or services, composite software systems can be modelled as a workflow connecting the processing elements. The workflow plays the role of the orchestration specifying the dependency constraints between the processing elements. In the workflow, each abstract processing element is defined by the functionality they must provide. When the software system is composed, concrete processing elements with the same functionalities are bound to the abstract elements. For each abstract element, several concrete elements may be available but only one must be picked up. What differs from these concrete elements is their QoS. The QoS of the whole workflow is obtained from its structure and from the individual QoS of the different concrete processing elements. An important problem is to be able to select the right concrete processing elements such that the resulting QoS of the provided service, i.e., the composition of concrete elements, satisfies some constraints and/or minimizes or maximizes the function of QoS attributes; see Cardoso (2002), Canfora (2005), Rosario *et al.* (2008), Zeng *et al.* (2004), Kelly *et al.* (2003), Fancsali (2003). For example, we may need a service that costs less than a given price and that returns the answer within a given amount of time. We may also want to get the cheapest service that answers within a given amount of time. The problem to find the solutions of this task is that the number of possible instantiations is exponential, i.e., a workflow with X abstract elements (e.g., $X = 10$) each having Y possible concrete elements (e.g., $Y = 10$) has X^Y (e.g., 10^{10}) possible concrete workflows.

The paper advocates to tackle this problem by means of the Constraint Logic Programming paradigm, using interesting properties of the search-space and (anti)-monotonic properties of the QoS attributes. Especially, the order in which elements are enumerated is crucial for efficiency reasons. To this end, we propose several enumeration strategies.

The rest of the paper is organized as follows. Section 2 defines the problem. Section 3 discusses the constraint logic programming paradigm and how it can be useful to solve the problem. Section 4 presents the proposed solution. Section 5 describes the results of the experimental research. Finally, Section 6 concludes the paper.

2. Problem Definition

2.1. *Processing Element Workflow*

The processing element workflow can be modelled by a set of connected abstract processing elements with defined functionalities. It represents a network of processing elements such that data enters a processing element through its incoming edges and leaves the element through its outgoing edges. Edges are connections between the output(s) of one processing element and the input(s) of another element. For abstraction purpose, the processing elements are black-boxes that compute outputs from inputs. Note that we consider And-split and And-join processing elements. And-split means that all the outgoing transitions are available after completing the execution of the processing element. And-join means that the processing element starts its execution when all its incoming transitions are enabled. This representation is the highest and simplest level of abstraction for component orchestration. To be more precise, the processing element workflow can be represented as a directed-acyclic graph (DAG) with loops which describes an ordering constraint between the processing elements. It contains only one start-node and one end-node representing respectively the input and the output of the whole workflow. Note that we allow loops, i.e., the processing element is performed a fixed number of times.

Concrete processing elements must be bound to the corresponding abstract processing elements that compose the workflow in order to instantiate the model and to provide the service associated with the workflow. For each abstract processing element, a number of concrete elements with the same functionality are available. Only the Quality of Service, provided by different processing elements, differs. A set of QoS attributes is attached to each concrete processing element. We consider, in the paper, three different types of Quality of Service attributes: cost, time, and availability, denoted respectively by $\mathrm{QoS_c}$, $\mathrm{QoS_t}$ and $\mathrm{QoS_a}$. The cost to be paid by the service user, if this element is employed, the time required to compute the output when the inputs are available, and the probability that the processing element is available when desired, is associated to each concrete element. Notice that, additional attributes can be employed provided that the value of the attribute for a composition of processing elements can be computed.

Figure 1 presents an example of workflow with five abstract processing elements $\{\mathrm{E_1}, \mathrm{E_2}, \mathrm{E_3}, \mathrm{E_4}, \mathrm{E_5}\}$ where $\mathrm{E_1}$ and $\mathrm{E_5}$ are, respectively, the start-node and the end-node. Table 1 presents examples of concrete processing elements with their associated quality of service that can be bound to the abstract elements of the workflow in the Fig. 1.

2.2. *Computing QoS Composition*

Let us now consider how to compute the QoS of a workflow, starting from the QoS attribute values of the processing elements. First, the QoS of the output of the start-node is computed, and then it is propagated to the inputs of the nodes that are connected to the start-node. If the QoS is available for all the inputs of a processing element then

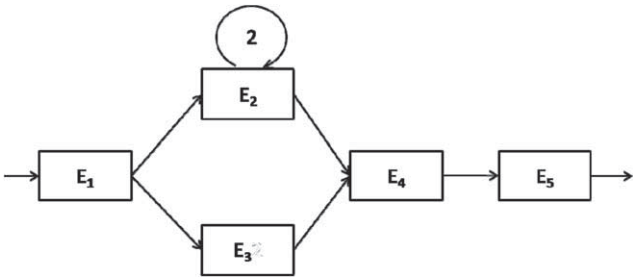

Fig. 1. Example of a processing element workflow.

Table 1
Example of concrete processing elements that can be bound to the abstract elements in Fig. 1

Concrete element	Can be bound to the abstract element	QoS_c	QoS_t	QoS_a
C_{11}	A_1	10	20	0.995
C_{12}	A_1	16	15	0.993
C_{21}	A_2	10	6	0.95
C_{22}	A_2	5	9	0.98
C_{31}	A_3	5	10	0.974
C_{32}	A_3	10	8	0.991
C_{41}	A_4	20	20	0.999
C_{42}	A_4	30	10	0.995
C_{51}	A_5	8	40	0.999
C_{52}	A_5	14	30	0.995

the QoS of the output of the processing element is computed. This process is performed recursively until computing the QoS of the output of the end-node which is the QoS of the whole workflow. How to compute the QoS of the output of a processing element from the QoS of its inputs and from its own QoS will be described bellow.

Figure 2 presents an abstract processing element E_i with the QoS attribute values QoS_{E_i} and the inputs $\{E_j, \ldots, E_I\}$ denoted as Inputs (E_i). Rule 1 specifies how to compute the QoS attribute values of the output of E_i denoted as Q_{E_i}.

Rule 1. The QoS attribute values of the output are computed as follows:

$$Q_{E_{i,cost}} = \sum_{k \in Inputs(E_i)} Q_{k,cost} + N \times QoS_{E_i,cost},$$

$$Q_{E_i,time} = \max_{k \in Inputs(E_i)} Q_{k,time} + N \times QoS_{E_i,time},$$

$$Q_{E_i,availability} = \prod_{k \in Inputs(E_i)} Q_{k,availabile} \times QoS^N_{E_i,availabile}.$$

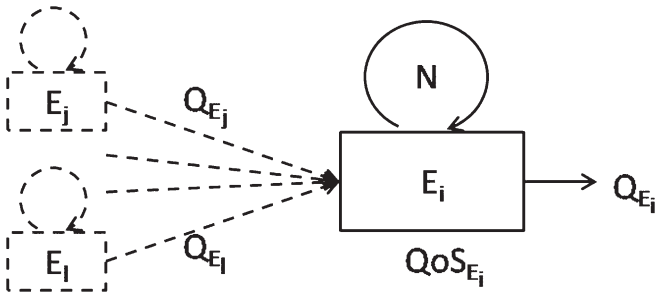

Fig. 2. Abstract processing element E_i.

2.3. *Problem Statement*

Given a processing element workflow and a list of concrete processing elements to be bound to the abstract elements, the problem of QoS-aware composition can be formulated with the following tasks:

Task 1. Identify what is the QoS of a given instantiation of the processing element workflow.

Task 2. Check, whether an instantiation of a processing element workflow does satisfy given QoS constraints.

Task 3. Find an instantiation of the processing element workflow that satisfies the given QoS constraints.

Task 4. Find all the instantiations of the processing element workflow that satisfy the given QoS constraints.

Task 5. Find an instantiation of the processing element workflow that optimizes (maximizes or minimizes) the function of QoS attributes and that satisfies the given QoS constraint.

Task 6. Find all the instantiations of the processing element workflow that optimize (maximize or minimize) the function of QoS attributes and that satisfy the given QoS constraint.

Referring to the example in Fig. 1 and Table 1, if we instantiate a workflow with the concrete elements [C_{12}, C_{21}, C_{32}, C_{41}, C_{52}], we obtain the Quality of Service: $QoS_c = 80$, $QoS_t = 77$ and $QoS_a = 0.838$ (Task 1). If we instantiate a workflow with the concrete elements [C_{12}, C_{22}, C_{32}, C_{42}, C_{51}], the concrete workflow satisfies the QoS constraint $QoS_c < 80$, $QoS_t < 100$ and $QoS_a > 0.8$ but not the constraint $QoS_c < 70$, $QoS_t < 80$ and $QoS_a > 0.95$, i.e., the Quality of Service of the concrete workflow is $QoS_c = 74$, $QoS_t = 83$, $QoS_a = 0.920$ (Task 2). Given the QoS constraint $QoS_c < 80$, $QoS_t < 80$ and $QoS_a > 0.9$, [C_{12}, C_{22}, C_{31}, C_{42}, C_{52}] is an instantiation that satisfies the QoS constraint (Task 3) and {[C_{12}, C_{22}, C_{31}, C_{42}, C_{52}], [C_{11}, C_{22}, C_{32}, C_{42}, C_{52}], [C_{11}, C_{22}, C_{31}, C_{42}, C_{52}]} is the set of all the instantiations that satisfy the QoS constraint (Task 4). If we want to minimize the function "$2 \times QoS_c + QoS_t$" while keeping QoS_a above 0.9 and QoS_t below 90, we can compute an optimal instantiation [C_{11}, C_{22}, C_{31}, C_{41}, C_{52}] where the minimal value of "$2 \times QoS_c + QoS_t$" is 206 (Task 5). With the function "QoS_c"

to minimize and the QoS constraints $\text{QoS}_\text{t} < 98$ and $\text{QoS}_\text{a} > 0.9$, we can compute all the optimal solutions {[C_{12}, C_{22}, C_{31}, C_{41}, C_{51}], [C_{11}, C_{22}, C_{31}, C_{41}, C_{52}]} where the minimal value of "QoS_c" is 59 (Task 6).

The problem we are facing is to find a computing paradigm for which all Tasks 1–6 can be easily formulated in a unified manner, so that they could be solved in an efficient way. The major problem is that the size of the search-space, i.e., the set of possible instantiations of the abstract workflow, can be huge. In a workflow with X abstract elements and Y concrete elements, which can be bound to each abstract element, the size of the search-space is X^Y. This is particularly problematic when looking for one or a few (optimal) solutions among a tremendous number of possible instantiations, which is like searching a needle in a haystack. We also want to be sure that the solution(s) of the tasks are correct and complete on the contrary to heuristics methods that seek to rapidly find a solution that is expected to be close to the good solution.

We argue that the Constraint Logic Programming paradigm (Colmerauer, 1987; Krzysztof, 2003) is a good candidate to solve Tasks 1–6 that are similar to discrete constraint satisfaction problems and optimization problems.

3. Constraint Logic Programming over Finite Domains

Constraint Logic Programming (Colmerauer, 1987; Krzysztof, 2003) is a paradigm that combines constraint programming and logic programming. A program consists of logic predicates that must be satisfied and that contain constraints in the body of predicates. Programs are queried about the satisfaction (in term of true/false)) of a goal that is defined by means of constraints and literals. The goal is proved if the bodies of the predicates are satisfied, i.e., constraints are consistent and literals are true using other predicates. This computing paradigm suits well for discrete optimization, constraint satisfaction and verification problems as scheduling, planning, packing and timetabling. The goal is to instantiate variables with the values taken from pre-defined domains such that the constraints are all satisfied.

A class of constraints used in Constraint Logic Programming is that of finite domains. The values of integer variables are taken from a finite domain. For each variable X of the problem, a domain is specified: X in [Min, Max] meaning that the value of X is an integer between Min and Max. The domain of a variable is reduced during the evaluation of queries. The program interpreter performs constraint propagations that enforce local consistency, that may reduce the domain of variables, e.g., if we have A in $[1, 4]$, B in $[2, 3]$ and A $>$ B, then the domain of A can be reduced to $[3, 4]$. If the domain of a variable becomes empty then the system is inconsistent, and the algorithm backtracks. If the domain of a variable contains only one value, then the variable can be assigned to this value. We use SWI-Prolog to define and interpret the queries and the "clpfd" library (Triska, 2010) to handle the constraints over finite domains.

A query in Constraint Logic Programming is usually formulated in the following way:

```
query(X):- constraints(X), labeling(X).
```

The predicate "query(X)" is true if the predicates "constraints(X)" and "labeling(X)" are true. The predicate "constraints(X)" adds the constraints of the CSP (constraint satisfaction problem) to the constraint store, i.e., defines the tree terms, real and finite domains variables of the system, fixes their domains and constrains the values of the variables w.r.t. the domain of the other variables (local and global constraints). The predicate "labeling(X)" performs a search over the domains of the variables of X to find an instantiation that satisfies all the constraints. Depending on the problem to be solved, only one or all of the correct instantiations of "X" can be computed. For example in Prolog, the predicate "findall$(\mathrm{X}, \mathrm{query}(\mathrm{X}), \mathrm{L})$" enables us to compute the list L of all valid instantiations of X that satisfy the predicate "query(X)". It is important to note that in Constraint Logic Programming, input parameters of predicates can be considered as "inputs" and "outputs" at the same time. In the "query(X)" predicate, the variable X can consist of several variables (tree term), e.g., $\mathrm{X} = [\mathrm{A}, \mathrm{B}, \mathrm{C}]$, such that when we ask for the satisfaction of the goal "query(X)", some variables of X are already instantiated (e.g., A and C as input variables) and some remain not-instantiated (e.g., B as an output variable). The interpreter of the query seeks an instantiation(s) of the not-instantiated variables that satisfy the constraints given the value of the instantiated variables.

Optimization problems can be solved using the goal "query(X)" taking into consideration the structure of the search-space and the properties of the function to be optimized.

This computing paradigm is convenient to solve Tasks 1–6. First of all, we need to constrain the QoS w.r.t. the structure of the abstract processing element workflow and the concrete processing elements to be bound. Thus, we can define the predicate "constraintsQoS$(\mathrm{APE}, \mathrm{CPE}, \mathrm{QoS})$" to be true if APE is the list of abstract elements of the workflow, CPE is a list of concrete processing elements and QoS is the QoS of the workflow whose possible values are constrained by the structure of the workflow and by the possible values of CPE. Finally, we need to define a predicate "labelingElement$(\mathrm{APE}, \mathrm{CPE}, \mathrm{QoS}, \mathrm{Option})$" to be true if:

- Option $=$ enum and each abstract element APE is instantiated with a concrete element of CPE (with the same functionality) or
- Option $= \mathrm{mia}(\mathrm{Mia}, \mathrm{Function})$, where Mia denotes "minimize" (Case 1) or "maximize" (Case 2), and each abstract element APE is instantiated with a concrete element of CPE (with the same functionality) such that the value of "Function" is minimized (Case 1) or maximized (Case 2).

We can now define the structure of our constraint logic program:

```
compositionQoS(APE, CPE, QoS, Option):-
    constraintsQoS(APE, CPE, QoS),
    labelingElement(APE, CPE, QoS, Option).
```

Referring to the example given in Fig. 1 and Table 1, Table 2 presents several examples of queries of the predicate "compositionQoS" for Tasks 1–6.

Table 2
Examples of queries to solve the tasks Tasks 1–6

Tasks	Query	Answer
Task 1	compositionQoS([C_{12}, C_{21},C_{32}, C_{41}, C_{52}], CPE, QoS, enum).	True, QoS = [80, 77, 0.838]
Task 2	QoS = [QoS_c, QoS_t, QoS_a], QoS_c<80, QoS_t<100, $QoS_a > 0.8$, compositionQoS([C_{12}, C_{22}, C_{32}, C_{42}, C_{51}], CPE, QoS, enum).	True, $QoS_c = 74$, $QoS_t = 83$, $QoS_a = 0.920$
Task 2	QoS = [QoS_c, QoS_t, QoS_a], QoS_c<70, QoS_t<80, $QoS_a > 0.95$, compositionQoS([C_{12}, C_{22}, C_{32}, C_{42}, C_{51}], CPE, QoS, enum).	False
Task 3	QoS = [QoS_c, QoS_t, QoS_a], QoS_c<80, QoS_t<80, $QoS_a > 0.9$, compositionQoS(APE, CPE, QoS, enum).	True, APE = [C_{12}, C_{22}, C_{31}, C_{42}, C_{52}], $QoS_c = 75$, $QoS_t = 73$, $QoS_a = 0.901$
Task 4	QoS = [QoS_c, QoS_t, QoS_a], QoS_c<80, QoS_t<80, $QoS_a > 0.9$, findall([APE, QoS], compositionQoS(APE, CPE, QoS, enum), L).	True, L = [[[C_{12}, C_{22}, C_{31}, C_{42}, C_{52}], [75, 73, 0.901]], [[C_{11}, C_{22}, C_{32}, C_{42}, C_{52}], [74, 78, 0.918]], [[C_{11}, C_{22}, C_{31}, C_{42}, C_{52}], [69, 78, 0.903]]]
Task 5	QoS = [QoS_c, QoS_t, QoS_a], QoS_c<80, QoS_t<80, $QoS_a > 0.9$, findall([APE, QoS], compositionQoS(APE, CPE, QoS, enum), L).	True, L = [[[C_{12}, C_{22}, C_{31}, C_{42}, C_{52}], [75, 73, 0.901]], [[C_{11}, C_{22}, C_{32}, C_{42}, C_{52}], [74, 78, 0.918]], [[C_{11}, C_{22}, C_{31}, C_{42}, C_{52}], [69, 78, 0.903]]]
Task 6	QoS = [QoS_c, QoS_t, QoS_a], $QoS_t < 98$, $QoS_a > 0.9$, findall(APE, compositionQoS (APE, CPE, QoS, mia(minimize, QoS_c)), L).	True, L = [[C_{12}, C_{22}, C_{31}, C_{41}, C_{51}], [C_{11}, C_{22}, C_{31}, C_{41}, C_{52}]], $QoS_c = 59$

4. QoS-Aware Composition

4.1. *Constraint Handling*

Now let us consider how to solve the Tasks 1–6 using constraints and to define the predicates "constraintsQoS" and "labelingConcreteElement", i.e., how to constrain the QoS variables w.r.t. the workflow structure and the concrete processing elements, and how to efficiently enumerate the concrete processing elements. To be able to find a first good solution efficiently that prunes, as soon as possible, irrelevant parts of the search-space, we need to exploit the properties of the search-space and that of the functions used to compute the QoS. To this end, we use Definition 1 and Property 1.

DEFINITION 1. A is monotonic (respectively anti-monotonic) with respect to B if and only if both A and B increase or decrease together (resp. if B increases then A decreases, and if B decreases then A increases).

Property 1. *The QoS attribute "Cost" of the output of a processing element is monotonic with respect to the QoS attribute "Cost" of the input and the QoS attribute "Cost" of the same element. The QoS attribute "Time" of the output of a processing element is monotonic with respect to the QoS attribute "Time" of the input, and the QoS attribute "Time" of the same element. The QoS attribute "Availability" of the output of a processing element is anti-monotonic with respect to the QoS attribute "Availability" of the input and the QoS attribute "Availability" of the same element.*

These properties are crucial to prune the search-space and then to render the tasks from Task 1 to Task 6 solvable in real-life situations. Indeed, we need to minimize the cost and time attributes, maximize the availability attribute and set constraints as $\mathrm{QoS_c} < \alpha$, $\mathrm{QoS_t} < \beta$ and $\mathrm{QoS_a} > \delta$. Once an abstract workflow is partially instantiated, we know that any additional instantiation will make increase (respectively decrease) the QoS value of the monotonic (resp. anti-monotonic) attributes of the whole workflow. It means that if a current workflow does not satisfy the constraint as $\mathrm{QoS_c} < \alpha$, $\mathrm{QoS_t} < \beta$ or $\mathrm{QoS_a} > \delta$, then the search-space can be safely pruned without losing any solution that satisfies the constraint.

To solve the optimization problem, we use the same properties. Once a valid solution is enumerated, a new constraint is added in the constraint store. While enumerating, we check if the partially-instantiated workflow can lead to a valid solution. In order to minimize (resp., maximize) a problem, we store in memory the smallest (resp. largest) value "M" of the variable "V" being minimized (resp., maximized) of the already enumerated workflows, and add the constraint V > M (resp. V < M). In the case where we want to extract all the optimal solutions, we add the constraint V >= M (resp. V =< M) instead. Thus, any monotonic QoS attribute that is required by the user to be as small as possible (e.g., cost and time) and any anti-monotonic QoS attribute that is required by the user to be as high as possible (e.g., availability) can be considered for QoS-aware component composition using our solution.

4.2. *Constraints for QoS Composition*

For the predicate "constraintsQoS", we propose the following structure:

```
constraintsQoS(APE, CPE, QoS):-
    initStartNode(APE),
    bounds(APE, CPE),
    dependencyconstraint(APE),
    endNode(QoS).
```

This predicate is true if the following predicates are true: "initStartNode", "bounds", "dependencyconstraint" and "endNode".

The predicate "initStartNode" simply sets that the QoS of the input of the start node is

$$\mathtt{QoS_c} = 0, \quad \mathtt{QoS_t} = 0 \quad \mathtt{and} \quad \mathtt{QoS_a} = 1.$$

The predicate "bounds" defines and instantiates the variables of the workflow. For each abstract processing element E and for each QoS attribute Q_j, two finite domains variables Q_E,Q_i and QoS_E,Q_i are defined. O_E, Q_i represents the value of the QoS attribute Q_j for the output of the processing element E_i. QoS_i, Q_E is the QoS value of E for the QoS attribute Q_i. The value of the variable O_E, Q_j is constrained by the type of QoS attribute considered (cost, time and availability) and the QoS values of the inputs of E. QoS_E, Q_i is constrained by the possible QoS values that can take concrete processing elements that can be bound by E. First of all, we compute the minimum and maximum values, denoted as MinE, Q_i and MaxE, Q_i respectively, of the QoS attribute Qj that can take concrete processing elements. Each variable QoS_EEQ_i is bounded by $[\mathrm{MinE}, Q_i, \mathrm{MaxE}, Q_i]$. Doing that, we limit the possible values of QoS_EQ_i and, by propagation; it also bounds the QoS of the whole workflow. Then, we compute the minimal and maximal differences between each pair of QoS attributes QoS_{E_k} and QoS_{E_I} of E denoted, respectively, as $\mathrm{MinDiff}_{k,j}$ and $\mathrm{MaxDiff}_{k,j}$. The constraints $QoS_{E_{Q_k}} - QoS_{E_{Q_l}} >= \mathrm{MinDiff}_{k,j}$ and $QoS_{E_{Q_k}} - QoS_{E_{Q_l}} <= \mathrm{MaxDiff}_{k,j}$ are added. This constraint itself does not reduce the bounds of the variables, but during the enumeration process additional propagations will be performed. In addition, we constrain the possible values of the n-tuples $[QoS_{E_{Q_1}}, \ldots, QoS_{E_{Q_n}}]$, i.e., $[QoS_{E_c}, QoS_{E_t}, QoS_{E_a}]$, to be an element of the n-tuples TCE where TCE refers to the list of QoS attribute values that can take the concrete processing elements. In our example (see Fig. 1 and Table 1), we have $\mathrm{TCEA1} = \{[10, 20, 0.995], [16, 15, 0.993]\}$. This constraint increases the propagation mechanisms, given that the QoS_E, Q_i variables are not only individually constrained to be bound in some domains by the structure of the workflow, but their possible values are also bounded by the values that take, or can take, the other QoS attribute variables. In our example, if we have the constrain $QoS_{A_{1_c}} > 12$, then thanks to this new constraint, the concrete processing element C_{12} will be automatically assigned to A_1 by propagation mechanisms without the need for enumeration. Using the "clpfd" library, this constraint can be set with the predicate tuples_in".

The predicate "dependencyconstraint" constrains the value of the QoS of the output of the processing elements Q_E,Q_i using Definition 1. This process is performed for each abstract processing element of the workflow. The predicate "endNode" simply specifies that the QoS of the whole workflow is equal to the QoS of the output of the end-node.

After doing that, all the constraints are set up with the following effects:

- The QoS of the whole workflow is constrained to be bound in an interval. In our running example, after setting the constraint, we obtain the following bounds for the QoS of the workflow: QoS_c in $[53, 90]$, QoS_t in $[67, 98]$ and QoS_a in $[0.82, 0.926]$.
- Any instantiation of an abstract processing element by a concrete one may lead to the reduction of the domain of the QoS and, in some situations, inconsistent constraint store and backtracking.

4.3. *Enumeration Strategy*

For the predicate "labelingElement", we propose the following structure:

```
labelingElement([], CPE, QoS, Option).
labelingElement(APE, CPE, QoS, Option) :-
    selectAbstractElement(APE, AE),
    selectConcreteElement(CPE, AE, CE),
    labeling(AE, CE),
    checkValidity(QoS, Option),
    labelingElement(APE\AE, CPE, QoS, Option).
```

This predicate is true if, either (1) no more abstract elements have to be instantiated, or (2) if the following three predicates are true: "selectAbstractElement", "labeling" and "checkValidity" and if the remaining abstract elements can be enumerated in a valid way.

The predicate "selectAbstractElement(APE, AE)" selects the next abstract element to be enumerated. The order in which elements are enumerated is crucial for efficiency reasons. To this end, we have developed several enumeration strategies. To select the right variable to be enumerated, we need to rank the different abstract processing elements according to the concrete processing elements that can be bound by it. In order to do that, for each processing element we first select the QoS attribute that will be used to perform the comparison. We propose the following eight different strategies to select the QoS attribute:

- MinInf: select a QoS attribute with the smallest lower-bound.
- MinSup: select a QoS attribute with the smallest upper-bound.
- MinInterval: select a QoS attribute having a tighter interval.
- MaxInf: select a QoS attribute with the highest lower-bound.
- MaxSup: select a QoS attribute with the highest upper-bound.
- MaxInterval: select a QoS attribute having a wider interval.
- Sum: create a new variable that is the sum of all the variables.
- Random: randomly select a QoS attribute.

Then, for each selected variable, i.e., the QoS attribute, we need to retrieve not only the interval of values but also the value. Therefore we propose four strategies:

- Inf: select the lower bound of the variable.
- Sup: select the upper-bound of the variable.
- Interval: select the size of the interval (upper-bound minus lower-bound).
- Mean: select the mean of the lower-bound and that of the upper-bound.

Finally, the abstract processing elements are ranked according to this value in an increasing (Inc1) or decreasing (Dec1) order. The predicate "selectAbstractElement(APE, AE)" is true if AE is the first abstract element in this ranking. Thus, 64 different enumeration strategies can be used to rank the abstract elements.

The predicate "selectConcreteElement$(\mathrm{CPE}, \mathrm{AE}, \mathrm{CE})$" is true if CE is a concrete element of CPE that has the same functionality as AE. Once again, to select the right CE to instantiate, we propose several strategies. We select the CE that has the smallest (SMa) or largest (LRg) QoS attribute value. Finally, the concrete elements are ranked in an increasing (Inc2) or decreasing (Dec2) order. We can select among 4 strategies to select the concrete elements. In total, we propose 256 enumeration strategies that will be evaluated in the experiments (see Section 5).

The predicate "labeling$(\mathrm{AE}, \mathrm{CE})$" is true if AE can be unified with CE, i.e., the abstract element AE is instantiated with the concrete element CE.

The predicate "checkValidity$(\mathrm{QOS}, \mathrm{Option})$" is true, if (1) Option=enum, or (2) mia$(\mathrm{Mia}, \mathrm{Function})$ and the upper-bound of Function is higher (if Option=maximize) or the lower-bound of Function is smaller (Option=minimize) than any valid corresponding QoS attribute of already completely instantiated workflow. After each instantiation, this predicate simply checks if the bounds of the QoS of the whole workflow can lead to a better solution or have to be pruned.

5. Experiments

5.1. *The Methodology of the Experimental Research*

Let us consider now the results of experiments performed on synthetic data-sets using the 2.4 GHz Intel Core Duo, 64-bit operating system, 4 GB of RAM, and Microsoft Windows Seven. We generated two abstract workflows with concrete processes with random QoS. We compare the various enumeration strategies for solving Task 5 and Task 6. First, we show (see Section 5.2) the number of enumerations necessary to solve the minimization Task without any QoS constraint with all the enumeration strategies with respect to the number of concrete processing elements (Task 5). We want to show the influence of the number of concrete processing elements and of the enumeration strategies on the number of instantiations (enumeration) required to solve the task. Second, we present experiments (see Section 5.3) on the minimization task with QoS constraints with respect to the parameter values in the QoS constraints (Task 6). We want to show how QoS constraints influence the difficulty to solve the problem.

The first dataset, denoted by D_1, is comprised of 13 abstract processes with sequences, loops, and parallel structures. The second data-set, denoted by D_2, is comprised of 10 abstract processes with sequences and parallel structures. For each point in the figures, we compute the average of the number of enumerations to solve a task for a set of 20 concrete processing elements generated with random QoS attribute values ($\mathrm{QoS_c}$ in $[1, 100]$, $\mathrm{QoS_t}$ in [1,100] and $\mathrm{QoS_a}$ in $[0, 1]$).

5.2. *Minimization Task Without the QoS Constraint*

For the first set of experiments, we consider Task 5, i.e., findall$([\mathrm{APE}, \mathrm{QoS}]$, composition QoS$(\mathrm{APE}, \mathrm{CPE}, \mathrm{QoS}, \mathrm{enum}), \mathrm{L})$ on the datasets D_1 and D_2. In Fig. 3, we present

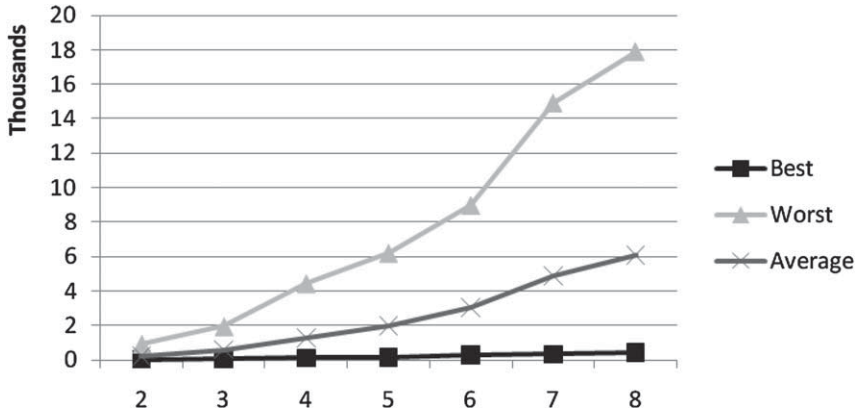

Fig. 3. Number of enumerations required to solve Task 5 w.r.t. the number of concrete elements for D_1.

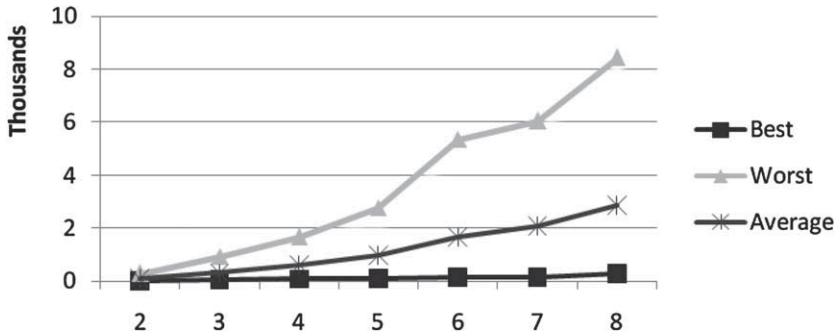

Fig. 4. Number of enumerations required to solve Task 5 w.r.t. the number of concrete elements for D_2.

the number of enumerations (Y axis) necessary to solve Task 5 for D_1 with the 256 enumeration strategies with respect to the number of concrete processing elements that can be bound on the abstract elements (X axis). The first curve "Best" shows the enumeration strategy that requires the least number of enumerations. The second curve "Worst" shows the worst enumeration strategy. The last curve "Mean" shows the average number of enumerations required by 256 enumeration strategies. Figure 4 shows the same results for the data-set D_2. For each figure, we performed 35,840 evaluations ($256 \times 20 \times 7$) of Task 5.

We can see in Figs. 3 and 4 that the enumeration strategy is crucial to efficiently prune the search-space and to quickly find a good solution. We have found out that the strategy (MinInf, Inf, Inc1, Min, SMa, Inc2) is the best enumeration strategy overall. In the remaining of the paper, this enumeration strategy will be employed. Figure 5 shows the number of enumerations required to solve Task 5 with the "best" enumeration strategy with respect to the number of concrete processing elements.

Figure 5 shows that, with the "best" enumeration strategy, Task 5 can be solved even if a lot of concrete processing elements are considered.

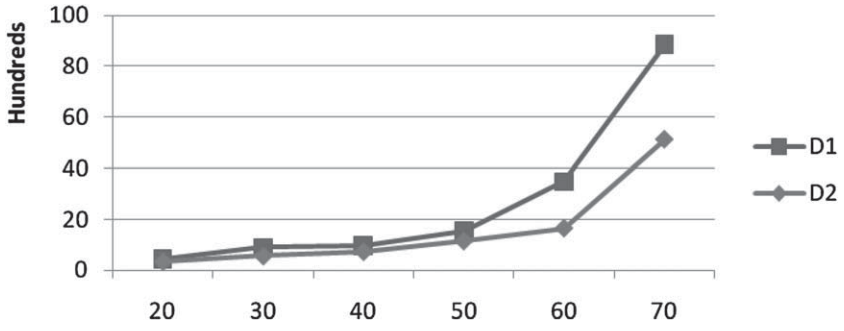

Fig. 5. Number of enumerations for the "best" strategy.

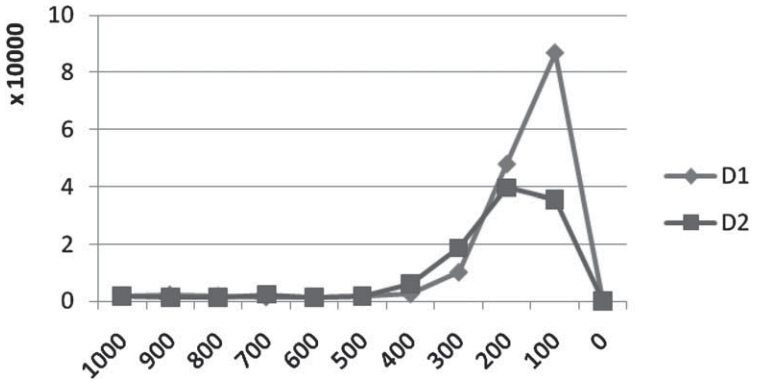

Fig. 6. Number of enumerations required to solve Task 6 w.r.t. the parameter value α for D_1 and D_2.

5.3. *Minimization Task Without QoS Constraint*

In the second set of experiments, we consider Task 6 on the data-sets D_1 and D_2, i.e., QoS = [QoS_c, QoS_t, QoS_a, $QoS_t < \alpha$, findall(APE, compositionQoS(APE, CPE, QoS, mia(minimize, QoS_c)), L). We want to estimate the influence of the QoS constraint parameter values. Figure 6 presents the number of enumerations (Y axis) required to solve Task 6 w.r.t. the QoS parameter value α (X axis).

Figure 6 shows that the number of enumerations required to solve Task 6 really depends on the QoS constraint parameters, i.e., the instantiation(s) with the lowest QoS_c value(s) must be selected among those that satisfy the QoS constraint.

6. Conclusion

The paper proposes to use Constraint Logic Programming to find the optimal QoS-aware composition of components or services that satisfies the given QoS constraints. This problem can be formulated as a discrete optimization problem. (Anti-)Monotonic properties of the search-space and of the QoS attributes are used to prune the search-space effi-

ciently and to quickly find a first good solution. The experiments show that the proposed approach performs well with artificial data and that it can be used to solved the QoS-aware composition problem. Comparing with the methods proposed by other authors to solve the problems discussed in this paper, the most important advantage of our approach is that, if the solution is tractable, the solution(s) of the tasks are correct and complete contrary to the heuristics methods that seek to rapidly find a solution that is expected to be close to the good solution. In addition, the method allows extracting only one optimal solution or the complete list of optimal ones. The last but not least, additional tasks such as finding the K-best solutions or those that are almost optimal, given a maximal error tolerance could be performed.

Acknowledgements. This work is partially funded by the Research Council of Lithuania under the grant No. MOS-4/2010.

References

BEA Systems, Inc., Cape Clear Software, International Business Machines Corp, Interface21, IONA Technologies, Oracle, Primeton Technologies, Progress Software, Red Hat, Rogue Wave Software, SAP AG., Siemens AG., Software AG., Sun Microsystems, Inc., Sybase Inc., TIBCO Software Inc. (2007a). *SCA Assembly Model Specification*, v1.0. March 2007. Available at: `http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf`.

BEA Systems, Inc., Cape Clear Software, International Business Machines Corp, Interface21, IONA Technologies, Oracle, Primeton Technologies, Progress Software, Red Hat, Rogue Wave Software, SAP AG., Siemens AG., Software AG., Sun Microsystems, Inc., Sybase Inc., TIBCO Software Inc. (2007b). *SCA Policy Framework*. Available at: `http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf`.

Canfora, G., Di Penta, M., Esposito, R., L.Villani, M. (2005). An approach for QoS-aware service composition based on genetic algorithms. In: Beyer, H. (Ed.). *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation. GECCO '05*. ACM, pp. 1069–1075.

Cardoso, J. (2002). *Quality of service and semantic composition of work flows*. PhD thesis, Univ. of Georgia.

Colmerauer, A. (1987). Opening the Prolog III universe. *BYTE* 12(9), 177–182. Available at: `http://portal.acm.org/citation.cfm?id=26023.26032`.

Fancsali, A. (2003). An extension of the Bellman–Ford algorithm for QoS routing with inaccurate information. *Informatica*, 27(4), 469–481.

Froulund, S., Koistinen, J. (1998). Quality of service specification in distributed object systems. *Distrib. Syst. Eng. J.*, 5(4), 179–202. Available at: `http://www.hpl.hp.com/techreports/98/HPL-98-159.pdf`.

Götze, J., Christiansen, P.E., Mortensen, R.K., Paszkowski, S. (2009). Cross-national interoperability and enterprise architecture. *Informatica*, 20(3), 369–396.

ISO/IEC 9126-1 (2001). *Software engineering – Product quality. Part 1. Quality model*. International Standard. ISO/IEC 9126-1:2001(E).

Kelly, B., Guy, M., James, H. (2003). Developing a quality culture for digital library programmes. *Informatica*, 27(3), 335–344.

Krzysztof, R. (2003). *Principles of Constraint Programming*. Cambridge University Press, Cambridge.

Kühne, S., Thränert, M., Speck, A. (2005). Towards a methodology for orchestration and validation of cooperative e-business components. In: Rutheford, J.M. (Ed.). *Proceedings of the 7th GPCE YRW*. Institute of Cybernetics, Tallinn Technical University, pp. 29–34. Available at: `http://www.ids-scheer.com/set/6591/KueTS05.pdf`.

Lamanna, D.D., Skene, J., Emmerich, W. (2003). SLAng: a language for service level agreements. In: *Proceedings of the 9th IEEE Workshop on Future Trends in Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, pp. 100–106.

Ludwig, H., Keller, A., Dan, A., King, R., Franck, R. (2003). *Web Service Level Agreement (WSLA) Language Specification*. IBM Corporation. Available at: http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf.

Mennie, D., Pagurek, B. (2000). An architecture to support dynamic composition of service components. In: Bosch, J., Szyperski, C., Weck, W. (Eds.). *Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000)*. Sophia Antipolis, France. Bleckinge Institute of Technology, Online publication. Available at: https://www.bth.se/fou/forskinfo.nsf/Sok/a6323cab891c823ec12569ba0048140c/$file/Research%20report%2015-00.pdf.

OASIS Consortium (2007). *Web Services Business Process Execution Language*, Version 2.0. OASIS Standard. Available at: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

OMG Consortium (2005). *UMLTM Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. OMG Standard. Available at: http://www.omg.org/docs/ptc/05-05-02.pdf.

Pal, P., Loyall, J., Schantz, R., Zinky, J., Shapiro, R., Megquier, J. (2000). Using QDL to specify QoS aware distributed (QuO) application configuration. In: *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2000)*. IEEE, pp. 310–319. Available at: http://csdl.computer.org/comp/proceedings/isorc/2000/0607/00/06070310abs.htm.

Plebani, P. (2006). Quality of Web services. In: *International Summer School on Service-Oriented Architectures*. Collegno (Chambery–Torino). Presentation slides. Available at: http://www.di.unito.it/~baroglio/SummerSchool06/Slides/QoWS.pdf.

Raje, R.R. (2000). UMM: unified meta-object model for open distributed systems. In: *Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000*, Hongkong, pp. 454–465. Avalaible at: http://www.cs.iupui.edu/uniFrame/pubs-openaccess/umm.pdf.

Rosario, S., Benveniste, A., Haar, S., Jard, C. (2008). Probabilistic QoS and soft contracts for transaction-based web services orchestrations. *IEEE Trans. Serv. Comput*., 1(14), 187–200. Accessible at: http://www.labri.fr/perso/anca/docflow/publications_files/QoSmonitoring.pdf.

Shaw, M. (1996). Truth vs. knowledge: the difference between what a component does and what we know it does. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. pp. 181–185.

TeleManagement Forum (2004). *SLA Management Handbook*. Vol. 4. Enterprise Perspective. The Open Group. Available at: http://www.afutt.org/Qostic/qostic1/SLA-DI-USG-TMF-060091-SLA_TMForum.pdf.

Triska, M. (2010). SWI-Prolog library: constraint logic programming over finite domains. In online *SWI-Prolog Reference Manual*. Available at: http://www.swi-prolog.org/man/clpfd.html.

W3C (2002). *Web Service Choreography Interface (WSCI) 1.0*. W3C Note 8 August 2002. Available at: http://www.w3.org/TR/wsci/.

W3C Consortium (2004). *OWL-S: Semantic Markup for Web Services*. W3C submission. Available at: http://www.w3.org/Submission/OWL-S/.

W3C (2004). *Web Services Architecture*. W3C Working Group Note 11 February 2. Available at: http://www.w3.org/TR/ws-arch/.

W3C (2005). *Web Services Choreography Description Language*, Version 1.0. W3C Candidate Recommendation 9 November 2005. Available at: http://www.w3.org/TR/ws-cdl-10/.

Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H. (2004). QoS-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5), 311–327.

J. Besson is a researcher in computer science working at the Vilnius University Institute of Informatics and Mathematics, Vilnius, Lithuania. His main research interests include data mining, bio-informatics, web service, and component composition.

A. Čaplinskas is a professor, principal researcher and the head of the Software Engineering Department at the Vilnius University Institute of Informatics and Mathematics, Vilnius, Lithuania. His main research interests include software engineering, information system engineering, legislative engineering, and knowledge-based systems.

Organizacijų informacinių sistemų komponentų kompozicijų, atsižvelgiančių į teikiamų paslaugų kokybę formavimas, panaudojant ribojimų logikos metodus

Jeremy BESSON, Albertas ČAPLINSKAS

Šiuolaikinės organizacijų integruotos informacinės sistemos žymia dalimi yra komponuojamos iš gatavų programinių komponentų bei iš per internetą pateikiamų kompiuterinių paslaugų. Tai daroma panaudojant vadinamąsias orkestravimo schemas, kuriose reikia atsižvelgti ne tik į tai, kad šitaip sukurta sistema turėtų visą reikiamą funkcionalumą, bet ir į tai, kad ji teiktų reikiamos kokybės paslaugas. Kitaip tariant, čia iškyla uždavinys kaip įvertinti paslaugų kompozicijų kokybę. Paprastai yra reikalaujama, kad kuriamoji sistema būtų optimali kai kurių paslaugų kokybės atributų požiūriu ir tuo pat metu kiti jos teikiamų paslaugų kokybės reikalavimai tenkintų bent jau tam tikrus minimalius reikalavimus. Straipsnyje pasiūlyta šį uždavinį traktuoti kaip diskrečiojo optimizavimo ir ribojimų tenkinimo uždavinį ir jį spręsti, panaudojant ribojimų logikos metodus.