A Constraint-Based Approach to Quality Assurance in Service Choreographies*

Dragan Ivanović,[1] Manuel Carro,[1,2] and Manuel Hermenegildo [1,2]

[1] School of Computer Science, T. University of Madrid (UPM), Spain
(idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es)
[2] IMDEA Software Institute, Spain

Abstract. Knowledge about the quality characteristics (QoS) of service compositions is crucial for determining their usability and economic value. Service quality is usually regulated using Service Level Agreements (SLA). While end-to-end SLAs are well suited for request-reply interactions, more complex, decentralized, multi-participant compositions (service choreographies) typically involve multiple message exchanges between stateful parties and the corresponding SLAs thus encompass several cooperating parties with interdependent QoS. The usual approaches to determining QoS ranges structurally (which are by construction easily composable) are not applicable in this scenario. Additionally, the intervening SLAs may depend on the exchanged data. We present an approach to data-aware QoS assurance in choreographies through the automatic derivation of composable QoS models from participant descriptions. Such models are based on a message typing system with size constraints and are derived using abstract interpretation. The models obtained have multiple uses including run-time prediction, adaptive participant selection, or design-time compliance checking. We also present an experimental evaluation and discuss the benefits of the proposed approach.

Keywords: Service Compositions, Quality of Service, Quality Assurance, Constraints, Abstract Interpretation.

1 Introduction

Service-Oriented Computing (SOC) is a widely-accepted paradigm for the development of highly dynamic, flexible, and distributed Service-Based Applications (SBAs). Service compositions allow putting together several specialized, loosely coupled, and platform-independent service components in order to perform complex and/or inter-organizational tasks [10]. In such scenarios, many of those components may be provided and controlled by third parties [22].

The Quality of Service (QoS) properties of service components and compositions are critical for their usability. Service Level Agreements (SLAs) are a means for defining permissible values for QoS attributes that are relevant in some scenario or for a particular purpose (such as execution time, monetary cost, or availability) and that a service (composition) provider is expected to deliver to a client. SLAs are commonly specified under the assumption that each interaction between the client and the service is viewed as a single session, and, accordingly, such end-to-end SLAs correspond to a request-reply message exchange pattern between the two parties. However, many business processes involve more complex message exchange patterns

* The authors were partially supported by Spanish MINECO project 2008-05624/TIN *DOVES* and Community of Madrid project P2009/TIC/1465 *PROMETIDOS-CM*.

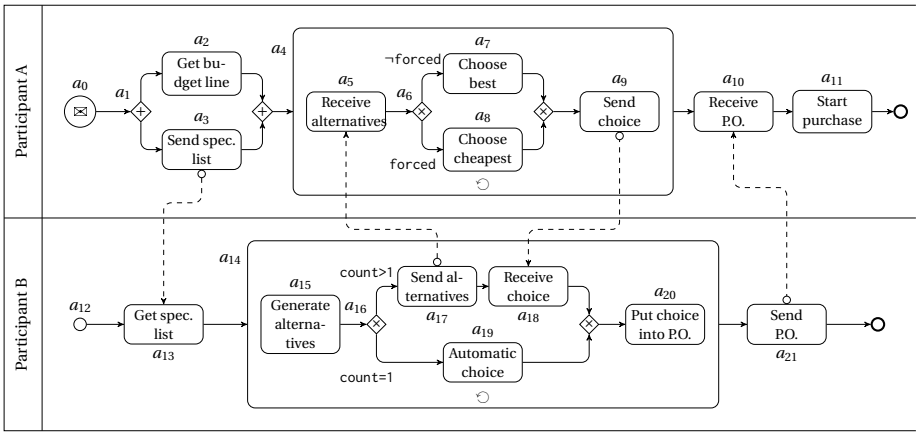

Fig. 1. An example choreography for purchase ordering.

between two or more stateful participants, where several interactions may belong to the same session and build upon each other, and where the data that is exchanged may significantly affect the behavior of the participants in terms of QoS, including the number of messages exchanged.

For such complex, multi-participant choreographies, a coherent support for QoS assurance which includes negotiation, prediction, and QoS-driven adaptation [16] is relevant both theoretically and practically. While several types of run-time adaptation aimed at avoiding or mitigating SLA violations have been proposed [10, 12, 21], these are often only applicable to the request-response message exchange pattern and/or to acyclic control structures. Several prediction and run-time adaptation approaches, more suited for orchestrations with centralized control flow, were proposed based on machine learning [15], online testing [19], and model checking [20].

In this paper, we propose a constraint-based approach for supporting QoS assurance for service choreographies that involve multiple, stateful participants and complex message exchanges. The proposed approach can be applied both at design time and at run time to support QoS negotiation, prediction, and QoS-driven adaptation. This work is an extension of [14] (on run-time prediction of SLA violations) for the case of service orchestrations with interconnected constraint models of stateful, interacting choreography participants, combining the derivation of QoS constraints with static analysis techniques.

We first present a motivational example (Section 2), then describe our approach (Section 3), review several examples of its application (Section 4), and finish with some conclusions (Section 5).

2 Motivation

Figure 1 shows a simplified example of a choreography for purchasing goods or services in a large organization where the procurement function is centralized. It uses the BPMN notation [17] with swim lanes delimiting participants, and dashed lines showing the flow of messages between them.

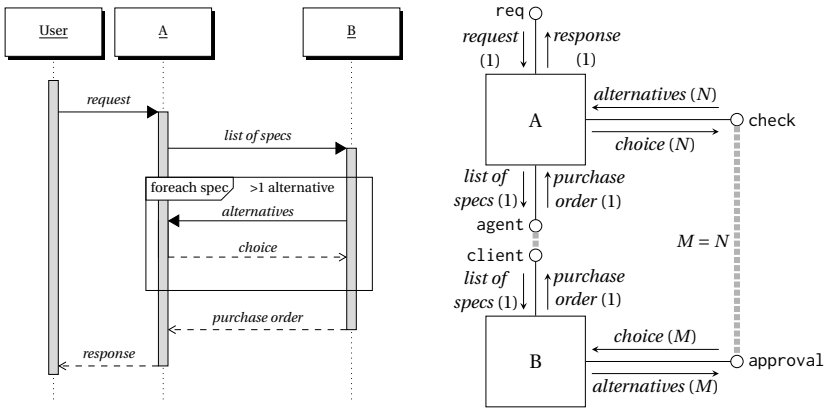

Fig. 2. Message exchange sequence and a component model for a choreography session.

Participant A is the procurement process, which starts by receiving a procurement request (a_0), and continues by sending the list of specifications to the agent (a_3) and retrieving budget line information for this purchase (a_2), in parallel. Participant B is the agent which receives the list of specifications (a_{13}) and performs a loop (a_{14}) for each item from the specification list. For each item, B looks into the supplier catalogs (a_{15}) to find out alternative purchasing options; since that can depend on the choice of earlier items, specifications are processed sequentially. If only one alternative is found, it is automatically chosen (a_{19}), but if two or more alternatives exist, B asks A to chose among them (a_{17}) and waits for the answer (a_{18}). The choice is added to the purchase order (a_{20}). After processing all specifications, agent B returns the final purchase order document to A (a_{21}). Whenever A is asked to choose between alternatives (a_5), it acts based on the budget line restrictions: if forced, it uses the cheapest option; otherwise it tries to chose the best solution. After answering all choice queries, A receives the purchase order from B (a_{10}) and starts the purchase (a_{11}), which provides the return notification to the requester, to whom the purchased goods and services will be delivered directly.

Figure 2 abstracts away the logic of the participants from Figure 1, and concentrates only on the exchange of messages. The left-hand side of the figure shows a sequence diagram for message exchanges in a session involving the initiating user and the participants A and B. The right-hand side of the figure shows A and B as components with connector links (`req`, `agent`, and `check` for A, `client` and `approval` for B), with messages sent and received over these links. The number of message of each kind within a single session is shown in parentheses. Wiring between the connectors is shown with thick dotted lines. For each wire, both the kind and the cardinality of messages in both directions must match.

The end-to-end QoS characteristics of A (such as, e.g., its execution time) depend on several factors. Firstly, if the number of specification items n is > 0, there can be between 0 and n callbacks from B to A in the *foreach* loop. Secondly, the behavior of A for each callback from B depends on whether it is forced to choose the cheapest alternative, which is known at the exit of a_2. Some of these factors are controlled by the user (n), some by third parties (a_2 which sets the `forced` flag for a_6, a_{15} which gen-

erates alternatives), and some on the implementation of A (the logic and complexity of determining the best choice in a_7). With respect to the quality assurance issues illustrated by this example, we are interested in tackling the following problems:

- *Automatically deriving a QoS model of the choreography for a given input request or a class of input requests.* Such a model can be used as an input for determining SLA offerings from the service provider to the users.
- *Using the QoS model of the choreography to predict SLA violations at run-time,* at different points in execution. E.g., greater accuracy of prediction can be obtained when the `forced` flag becomes known after a_2.
- *SLA compliance checking of choreography participants at design-time for a given class of input requests.* This is the basis for adaptive dynamic selection (binding) of service components.

3 Constraint-Based QoS Modeling for Choreographies

The proposed constraint-based approach to modeling QoS for service choreographies is implemented in two principal phases. The first one focuses on the creation of the QoS models for the choreography participants as Constraint Satisfaction Problems [7, 1] (CSP). We will show how to generate a model of the QoS metrics under consideration (Section 3.1), capture the view of each participant regarding the effective QoS at every moment in the execution (Section 3.2), and how to automatically derive these models (Section 3.3). The model is enriched with information about the shape and size of messages, inferred using static analysis techniques, in order to increment its accuracy and the precision of the prediction (Section 3.4).

n.0: MH: This paragraph is difficult to understand?
n.1: DI, MCL: better now?

[0] [1] The second phase of the approach consists of connecting the models for the different participants and solving them as a whole (Section 3.5). Note that when deriving QoS models for choreographies, joining the different sub-models is done following the structure of the composition. In the present case, the overall structure may not lend itself to structural analysis and participants take a prominent role. Therefore determining the overall QoS characteristics is done by joining per-partner models (Section 3.5) mimicking the topology of the choreography.

3.1 Modeling Cumulative QoS Metrics

Execution time, availability, reputation, bandwidth consumed, and cost are some of the most common QoS attributes. In this work we focus on attributes that can be numerically quantified using some measurement scale, or QoS metric: e.g., execution time can be measured using time units. QoS metrics do not need to have a fixed origin (a "true zero" value), but one unit of distance needs to express the same variation in the attribute everywhere on the scale. This requirement excludes, for instance, ordinal voting-based reputation ranking between services, where the unit difference in ranking does not carry information about the difference in votes received.

We additionally require QoS metrics to be cumulative and non-negative: QoS values of activities in a sequence add up to give the QoS value for the sequence, and this value should never decrease by adding more activities. Some QoS metrics, such as

availability (expressed in terms of probabilities), that do not use addition to calculate aggregation in a sequence, can be converted into additive metrics using a suitable transformation. For instance, the availability p of n sequential activities with availabilities p_i is $p = \Pi_{i=1}^{n} p_i$ and can be converted into $\lambda = \Sigma_{i=1}^{n} \lambda_i$ with the transformation $\lambda = -\log p$.

Cumulative QoS metrics allow us to represent the QoS of a service composition at any point in execution as a sum of two components: the *previously accumulated* QoS up to that point, and the *pending* QoS for the remainder of the execution. Non-negativity guarantees that the pending QoS can only decrease as the execution proceeds. While the accumulated QoS can be estimated empirically (by measuring elapsed time, network traffic, or accumulated monetary cost), the pending QoS for the remainder of the execution is in our approach modeled as a CSP over variables that represent QoS values for composition activities and control constructs. Solving this CSP gives a prediction of the pending QoS.

3.2 QoS Models of Participants and Continuations

Service choreographies provide a "global view" of a multi-participant, stateful message exchange within some logical unit of work. There are several possibilities to provide both abstract and executable descriptions of choreographies. On the more abstract side BPMN (as in Figure 1) or WS-CDL [24], which is a high-level specialized choreography language, can be used. On the more executable side, we can use choreography extensions of standard process (orchestration) languages, such as BPEL4Chor [8]. In our approach, we assume that the implementation details of the participants are essentially private and that the participants can be viewed as communicating components that conform to the protocol (as in Figure 2). Conformance, compatibility, and realizability of choreographies has been studied using formal methods such as Petri Nets [23], session types [9], and state machines [3].

As mentioned before, we proceed by developing a separate QoS model for each participant in the choreography. Each participant is seen as a component with a number of connector links (or channels, in WS-CDL terminology). Each link c is bi-directional, and each direction (in/out) is characterized by a triplet of the form $\langle N_{\text{in/out}}(c), \bar{q}_{\text{in/out}}(c), \Delta\bar{q}_{\text{in/out}}(c)\rangle$, where N is multiplicity of in/out messages, $\bar{q}$ are QoS values corresponding to the first in/out message, and $\Delta\bar{q}$ are increments of QoS values for the successive messages (for N>1). For example, for the case of execution time, $T_{\text{in}}(\mathsf{c})$ is the time when the first message was received over link c and $\Delta T_{\text{in}}(\mathsf{c})$ is the time interval between the successive messages. N, $\bar{q}$ and $\Delta\bar{q}$, as well as other variables in the constraint QoS models developed in this section, are not numeric constants, but represent intervals of possible numeric values for all legal execution cases, whose upper and lower bounds are inferred from the constraint model.

We build the QoS model of a participant by looking at its current point in execution. To stay close to the executable specifications, we follow the same approach as in our previous work on run-time prediction for orchestrations [14]. We use the notion of a *continuation* which describes the current state of the participant and the remainder of the computation until its end [18]. At the beginning, the continuation is the entire process and it is gradually reduced by eliminating the completed activities

$$
\begin{array}{rll}
S := & \texttt{send}(c, E) \mid \texttt{recv}(c, v) \mid \texttt{invoke}(c, E, v) & (\textit{send/receive messages}) \\
\mid & \texttt{let}\ v = E & (\textit{variable assignment}) \\
\mid & [S, S, ..., S] & (\textit{sequence of } n \geq 0 \textit{ activities}) \\
\mid & (\texttt{if}(E) \rightarrow S; S) & (\textit{if-then-else}) \\
\mid & S\ \texttt{and}\ S & (\textit{parallel "and" split/join}) \\
\mid & \texttt{foreach}(v : E)\ \texttt{do}\ S & (\textit{iterate over list elements}) \\
\mid & \texttt{foreach}(\texttt{recv}(c, v))\ \texttt{do}\ S & (\textit{iteratively receive multiple messages}) \\
\mid & \texttt{stream}(c)\ \texttt{do}\ S & (\textit{send multiple messages}) \\
\mid & \texttt{relax} & (\textit{do nothing}) \\
c, v := & \langle \textit{identifier} \rangle & \\
E := & \langle \textit{expression} \rangle &
\end{array}
$$

Fig. 3. Abstract syntax of the participant continuation language.

as the execution proceeds. The continuation information is always implicitly present in the state of the engine which executes the participant, and, in principle, it can be obtained either by inspecting its internal state or by observing the process events from the outside. The latter is less robust since missed events or run-time service modifications can invalidate the information inferred through external observation.

We represent continuations using an abstract language for the participant processes (Figure 3). It is based on a prototypical process language implementation that provides the continuation information explicitly at each execution step [14]. The participant state is kept in variables whose types are described in Section 3.4. Variable values are assigned using the `let` construct or received over some link with `recv`. The standard sequential operator, *if-then-else*s, and AND-parallel splits/joins are supported. For simplicity, we present only two `foreach` looping constructs: one over elements of a list and another one over messages received over some channel. The `send` and `recv` messaging constructs can be combined into an `invoke`; note that *request-reply* patterns are not enforced (this is left to the protocol). Participants use the `stream` construct to send a series of messages within the same session which can be received with a `recv`-based `foreach`.

3.3 Automatic Derivation of the QoS Constraint Model for a Participant

The constraint QoS model for a participant is derived automatically from the continuation and the previously accumulated QoS, using the structural approach [14], where QoS values for complex constructs are constructed from their structural components. A separate constraint QoS model is derived for each QoS metric of interest. Due to space constraints, we will present here only on the derivation of execution time. The reader is kindly referred to our previous publication [14] for more detailed explanations and treatment of other metrics, such as availability.

Figure 4 shows the automatically derived QoS constraint model for the execution time for participant A, at its start, i.e., when the continuation consists of the entire participant process. The code for the participant A is shown on the left-hand side in the abstract syntax, and the generated constraints appear on the corresponding lines

1	`recv(req,request),`	$T_1^- = \max(T_A, T_{\text{in}}(\texttt{request}))$, $T_4^+ = T_1^-$, $N_{\text{in}}(\texttt{req}) = 1$;
2	`( invoke(budget,`	$T_4^+ \le T_2^+ \le T_4^-$, $T_{\text{out}}(\texttt{budget}) = T_2^+ + t_{\text{send}}$,
	`request,line)`	$T_2^- = T_{\text{in}}(\texttt{budget})$, $N_{\text{in/out}}(\texttt{budget}) = 1$
3	`and send(agent,`	$T_4^+ \le T_3^+ \le T_4^-$,
	`request/specs)`	$T_3^- = T_{\text{out}}(\texttt{agent}) = T_3^+ + t_{\text{send}}$, $N_{\text{out}}(\texttt{agent}) = 1$
4	`),`	$\max(T_2^- - T_2^+, T_3^- - T_3^+) \le T_4^- - T_4^+ \le (T_2^- - T_2^+) + (T_3^- - T_3^+)$
5	`foreach(recv(check,alts)) do`	$T_5^+ = \max(T_4^-, T_{\text{in}}(\texttt{check}),)$ $k_5 = N_{\text{in}}(\texttt{check}) \ge 0$
6	`[ ( if(not(line/forced))`	$T_6^+ = T_7^+ = T_8^+$, $c_6 \in \{0, 1\}$, $L_5 = \max(T_{10}^- - T_6^+, \Delta T_{\text{in}}(\texttt{check}))$
7	`-> invoke(best,alts,choice)`	$T_7^- = T_7^+ + \Delta T_{\text{best}}$
8	`; let choice=first(alts)`	$T_8^- = T_8^+ + t_{\text{expr}}$
9	`),`	$(c_6 = 1 \wedge T_6^- = T_7^-) \vee (c_6 = 0 \wedge T_6^- = T_7^-)$
10	`send(check,choice)`	$N_{\text{out}}(\texttt{check}) = k_5$, $T_{\text{out}}(\texttt{check}) = T_{10}^+ = T_6^-$, $\Delta T_{\text{out}}(\texttt{check}) = L_5$
11	`],`	$T_5^- = T_5^+ + k_5 \times L_5$
12	`recv(agent,po),`	$T_{12}^- = \max(T_5^-, T_{\text{in}}(\texttt{agent}))$, $N_{\text{in}}(\texttt{agent}) = 1$
13	`send(req,po)`	$T_{13}^+ = T_{12}^-$, $N_{\text{out}}(\texttt{req}) = 1$, $T_{13}^- = T_{\text{out}}(\texttt{req}) = T_{13}^+ + t_{\text{send}}$

Fig. 4. Structurally derived QoS constraint model for participant A.

to the right. For an activity on line i, we mark its starting time with T_i^+ and its end time with $T_i^-, T_i^- \ge T_i^+$. T_A represents the execution time at the current execution point (here at the start), and is an input to the model.[3] The code communicates over channels `req`, `agent`, and `check` from Figure 2, plus an additional channel `budget` which is used to invoke the budget line information service a_2 from Figure 1.

The execution of participant A is a sequence of commands, and the metric for the execution time is cumulative, for a sequence $S = [S_1, S_2, ..., S_n]$ we have $T^+ = T_1^+$, $T^- = T_n^-$, and for adjacent activities S_i and S_{i+1} we have $T_i^- = T_{i+1}^+$. For clarity of presentation, here we ignore the internal time used by the process engine between steps, which needs to be taken into account in real applications (see [14]).

The reception of a single message with $\texttt{recv}(c, v)$ (lines 1 and 12) finishes at time $T_i^- = \max(T_j^-, T_{\text{in}}(c))$, where T_j^- is the finish time of the previous activity, and $T_{\text{in}}(c)$ is the time at which the message arrives on the channel c. Since in our case messages are received over the same channel at a single place in code, the `recv` construct also sets $N_{\text{in}}(c) = 1$. The command $\texttt{send}(c, E)$ (lines 3, 10, 13) delivers a message to the mailbox on the other side of the channel, for which it takes some time marked with t_{send}, which is also a constrained variable and considered an input to the model. $T_{\text{out}}(c)$ is equated with the finish time T_i^- of the `send` construct. Outside a loop (lines 3 and 12), $N_{\text{out}}(c)$ is set to 1, and $\Delta T_{\text{out}}(c)$ is not constrained, because it is not applicable. The `invoke` construct in line 2 is treated as a `send`-`recv` sequence.

The timing for the AND-parallel flow (ending in line 4) depends on the particular process engine implementation, and can vary between real parallelism and sequential execution of the two activities. Without a more detailed knowledge of the implementation details, the duration of the parallel flow $T_4^- - T_4^+$ may vary between the maximum and the sum of durations of the two "parallel" activities.

The `recv`-based loop (line 5) starts when both the preceding activity has finished (T_4^-) and the first message on the `check` channel has become available ($T_{\text{in}}(\texttt{check})$).

[3] Remember (Section 3.2) that these variables actually contain admissible ranges.

$$
\begin{array}{lr}
\tau := \texttt{any} \mid \texttt{none} & (\textit{some unspecified value and no value}) \\
\mid \texttt{bool}(a..b) & (\textit{Boolean between } a \textit{ and } b,\ a,b \in \{0,1\},\ a \le b) \\
\mid \texttt{number}(a..b) & (\textit{number between } a \in \mathbb{R} \cup \{-\infty\} \textit{ and } b \in \mathbb{R} \cup \{+\infty\},\ a \le b) \\
\mid \texttt{string}(a..b) & (\textit{string with finite size between } a \in \mathbb{N} \textit{ and } b \in \mathbb{N} \cup \{+\infty\},\ a \le b) \\
\mid \texttt{list}(a..b, \tau) & (\textit{list with finite size between } a \in \mathbb{N} \textit{ and } b \in \mathbb{N} \cup \{+\infty\},\ a \le b) \\
\mid \{x_1 : \tau, x_2 : \tau, ..., x_n : \tau\} & (\textit{record with named fields } x_1, ..., x_n,\ n \ge 0)
\end{array}
$$

Abbrev.: $\texttt{bool} \equiv \texttt{bool}(0..1)$, $\texttt{number} \equiv \texttt{number}(-\infty..+\infty)$, $\texttt{string} \equiv \texttt{string}(0..+\infty)$, $\texttt{list}(\tau) \equiv \texttt{list}(0..+\infty, \tau)$

Fig. 5. A simple typing system for messages with size constraints

The number of iterations of the loop k_5 equals the number of messages arriving through the channel, $N_{\text{in}}(\texttt{check})$. Since every loop iteration can start only upon message reception, the effective length of a loop iteration L_5 is the maximum between the actual duration of the loop iteration ($T_{10}^- - T_6^+$) and the interval between incoming messages $\Delta T_{\text{in}}(\texttt{check})$. Sending a message in each iteration of the loop (line 10) equates the multiplicity of outgoing messages $N_{\text{out}}(\texttt{check})$ to the number of loop iterations k_5, and the interval between messages $\Delta T_{\text{out}}(\texttt{check})$ to the effective iteration length L_5.The *if-then-else* construct (line 6) introduces a binary constraint variable c_6 which captures the truth value of the condition, and a disjunctive constraint (line 9) which covers the *then* and the *else* cases. Finally, the internal operations, such as the expression evaluation (line 8) and a call to an internal procedure `best` (line 7), simply add the corresponding time intervals (resp. Δt_{expr} and Δt_{best}).

3.4 Analysis of Message Types With Size Constraints

The constraint QoS models whose derivation we described above include a number of internal structural parameters, such as the number of loop iterations and condition truth values (k_5 and c_6 in Figure 4) that depend on data that is received by these services. There are several ways in which the information about shape of the data can be organized and used to further constrain the values of these structural parameters and, therefore, make the constraint models more precise. One possibility would be to apply *computational cost analysis* techniques to an appropriate abstraction of the participant processes in order to obtain an analytic functional relationship between the size of input data (number magnitudes, list lengths, etc.) and the upper and lower bounds of possible values for the structural parameters [13]. Another possibility, which we discuss in this subsection, is to use a simple form of type analysis which is directly applicable to the abstract representations of continuations used in our approach.

Figure 5 shows a simple type system with size constraints, which includes Booleans, numbers, strings, lists, and records with named fields. Each type τ in this system has its denotation $[\![\tau]\!]$ which is the set of all values that belong to it. For instance, $[\![\texttt{number}(0..1)]\!] = \{x \in \mathbb{R} \mid 0 \le x \le 1\}$. By definition, we take $[\![\texttt{none}]\!] = \emptyset$. We write $\tau_1 \sqsubseteq \tau_2$ as a synonym for set inclusion $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. The set of all types with size constraints together with the relation $\sqsubseteq$ forms a *complete lattice* [6] with any as the top element, and none as the bottom element, i.e., $\texttt{none} \sqsubseteq \tau \sqsubseteq \texttt{any}$ for arbitrary τ. We

1	`recv(client,specs),`	$\tau_{\text{in}}(\texttt{client}) = \texttt{list}(a..b, \tau_{\text{spec}})$, $1 \le a \le b$
2	`let po = [],`	$\texttt{po} : \texttt{list}(0..0, \texttt{none})$
3	`stream(approval) do`	
4	`  foreach(spec:specs) do [`	$a \le k_4 \le b$
5	`  invoke(gen,spec,alts),`	$\tau_{\text{out}}(\texttt{gen}) = \tau_{\text{spec}}$, $\tau_{\text{in}}(\texttt{gen}) = \texttt{list}(1..+\infty, \tau_{\text{alt}})$
6	`  ( if(count(alts)>1)`	
7	`  -> invoke(approval, alts,choice)`	$\tau_{\text{out}}(\texttt{approval}) = \texttt{list}(1..+\infty, \tau_{\text{alt}})$, $0 \le N_{\text{out}}(\texttt{approval}) \le \max(a,b)$
8	`  ; let choice=first(alts)`	
9	`  ),`	$\texttt{choice} : \tau_{\text{alt}}$
10	`  let po = po + [choice]`	$\texttt{po}_{\text{before}} : \texttt{list}(n..m, \tau) \Rightarrow$ $\Rightarrow \texttt{po}_{\text{after}} : \texttt{list}(n+1..m+1, \tau \sqcup \tau_{\text{alt}})$
11	`  ],`	$\texttt{po} : \texttt{list}(a..b, \tau_{\text{alt}})$
12	`send(client,po)`	$\tau_{\text{out}}(\texttt{client}) = \texttt{list}(a..b, \tau_{\text{alt}})$

Fig. 6. Analysis of types with size constraints for participant B.

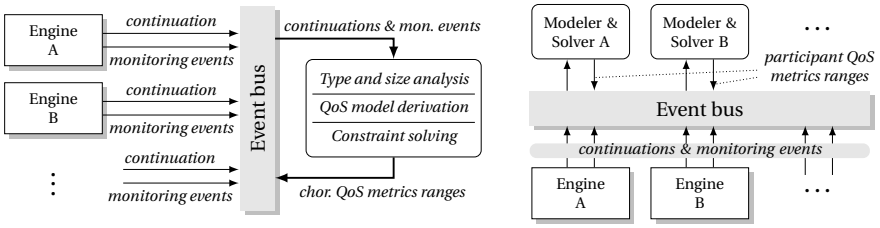

Fig. 7. Centralized (left) and distributed (right) processing of choreography QoS constraints.

introduce the *least upper bound* operation $\sqcup$ on types, where $\tau_1 \sqcup \tau_2 = \tau$ means that τ is the smallest type (w.r.t. $\sqsubseteq$) such that $\tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$. For example, $\texttt{number}(0..10) \sqcup \texttt{number}(8..100) = \texttt{number}(0..100)$, $\texttt{list}(1..5,\texttt{number}) \sqcup \texttt{list}(9..9,\texttt{bool}) = \texttt{list}(1..9,\texttt{any})$, and $\texttt{none} \sqcup \tau = \tau \sqcup \texttt{none} = \tau$.

The lattice structure of types from Figure 5 provides a domain for the application of abstract interpretation-based analysis techniques [4] to obtain a combination of type and size analysis for data in the participant processes before constructing the QoS model. This kind of analysis is well suited for our case in which looping is done by iterating over list elements and streams of messages, where the size range of the list type directly translates into the range of loop iterations. We enrich the link (channel) descriptions by adding input and output message types, $\tau_{\text{in}}(c)$ and $\tau_{\text{out}}(c)$.

For instance, in Figure 4, we start with $\tau_{\text{in}}(\texttt{req}) = \{\texttt{specs} : \texttt{list}(a..b, \tau_{\text{spec}}), \texttt{userId} : \texttt{number}\}$ where $a \ge 1$ and we derive that $\tau_{\text{out}}(\texttt{budget}) = \texttt{number}$ and $\tau_{\text{in}}(\texttt{budget}) \sqsubseteq \{\texttt{forced} : \texttt{bool}\}$. Also, in participant B, $\tau_{\text{out}}(\texttt{agent}) = \texttt{list}(a..b, \tau_{\text{spec}}) = \tau_{\text{in}}(\texttt{client})$. The result of the analysis for B is shown in Figure 6. From it, we infer that A.$N_{\text{in}}(\texttt{check})$ = B.$N_{\text{out}}(\texttt{approval})$ is between 0 and max(a, b).

3.5 Centralized and Distributed Processing of QoS Constraints

Solving a constraint model involves finding the sets of values for the constrained variables that satisfy the set of constraints, or signaling that the set of constraints is

inconsistent, and therefore unsatisfiable. Constraint solvers sometimes need to give an approximation of the actual solutions. These approximations are always complete (no solution is discarded), but maybe not correct (they may contain values that are not part of any solution [7]). Some constraint solvers are better suited for some constraint domains and classes of constraints than others. E.g., if the generated constraints are linear, a linear constraint solver is likely to detect inconsistencies and to narrow down the value sets closer to the actual answers, compared to a more general one. The constraint models generated using our approach in general involve nonlinear integer and real arithmetic constraints, as well as disjunctive constraints.

The constraint QoS models for each participant can be, in principle, derived and analyzed for the different message types separately, and the models obtained in that way can be composed together by connecting the appropriate input/output links and solving the resulting integrated model centrally. This architecture is shown on the left-hand side of Figure 7. Different participants may, in general, execute on different nodes (process execution engines) in a Service-Oriented System (SOC). They publish participant continuations and the related monitoring events (which can be used for establishing the previously accumulated QoS) to an event bus. An aggregated feed of continuations is read from the event bus and processed by a single component that performs the analysis, modeling, and constraint solving of the integrated participant models, and publishes the (updated) QoS metrics ranges for the entire choreography. An advantage of the centralized approach is that it offers integrated information about the behavior of the participants and QoS for the choreography. However, it may not scale well, since it requires global streaming of continuations, monitoring events and results to and from a single processing component. Besides, it can be undesirable in some settings since data regarding execution characteristics may need to be sent out of their respective administrative domains to a central point.

A decentralized approach aimed at alleviating somehow these issues is shown on the right-hand side of Figure 7. Here, continuations and monitoring events published by process engines are processed by modules which can be close in the network topology to the engines, and inside their administrative boundaries. These modules perform a per-participant QoS analysis that updates the ranges for $\langle \tau_{\text{out}}, N_{\text{out}}, \bar{q}_{\text{out}}, \Delta\bar{q}_{\text{out}} \rangle$ for each outgoing channel using the corresponding ranges for $\langle \tau_{\text{in}}, N_{\text{in}}, \bar{q}_{\text{in}}, \Delta\bar{q}_{\text{in}} \rangle$ that are produced by the modelers/solvers for participants at the other end. The updates are communicated to the connected participant models and the process is repeated until a stable solution is reached. This can be achieved using distributed constraint solving algorithms [11], which ensure termination, completeness, and correctness.

4 Examples of Application

In this section, we illustrate how the proposed constraint-based approach can be of benefit in providing answers to the questions posed at the end of Section 2, using the motivating example. The idea of the approach is to be fully automated and supported by tools. Our current prototype executes processes written in the continuation language (Section 3.2), transmits continuations, and formulates and solves the QoS constraint models.

Ranges for internal activity parameters									
Parameter name	Confidence interval 99% parameter range [ms]			Confidence interval 90% parameter range [ms]			Confidence interval 80% parameter range [ms]		
a_3: t_{budget}	500	..	1 500	642	..	1 167	673	..	1 094
a_7: t_{best}	100	..	700	195	..	509	215	..	468
a_{15}: t_{gen}	200	..	500	247	..	404	257	..	384
t_{send}	25	..	150						

Case 1: Varying confidence intervals for participants A and B									
Spec. list size	Confidence interval 99% T_{out}(req) range [ms]			Confidence interval 90% T_{out}(req) range [ms]			Confidence interval 80% T_{out}(req) range [ms]		
1 .. 10	274	..	17 100	322	..	14 868	332	..	14 376
11 .. 20	2 274	..	32 100	2 797	..	27 970	2 912	..	27 057
21 .. 50	4 274	..	77 100	5 272	..	67 273	5 492	..	65 103
50 .. 100	10 074	..	152 100	12 450	..	132 780	12 972	..	128 512
101 .. 200	20 274	..	302 101	25 069	..	263 793	26 128	..	255 330

Case 2: Varying confidence intervals for A and B with `force=true`									
Spec. list size	Confidence interval 99% T_{out}(req) range [ms]			Confidence interval 90% T_{out}(req) range [ms]			Confidence interval 80% T_{out}(req) range [ms]		
1 .. 10	274	..	10 100	322	..	8817	332	..	8 535
11 .. 20	2 274	..	18 100	2 797	..	15 867	2 912	..	15 376
21 .. 50	4 274	..	42 100	5 272	..	37 017	5 492	..	35 900
50 .. 100	10 074	..	82 100	12 450	..	72 268	12 972	..	70 106
101 .. 200	20 274	..	162 100	25 069	..	142 768	26 128	..	138 518

Table 1. Experimental inputs and outputs of the execution time model.

4.1 Supporting SLA Negotiation For Classes of Input Data

A constraint-based QoS model can be used at design time to help the providers of the participating processes in a choreography develop realistic SLA offers that can be used to negotiate with their users. In such a case, participant providers (e.g., the provider for participant A from Figure 1) can use the derived models, along with assumptions and empirical assessments of the behavior of the environment (network latency, component behavior, etc.) to develop reasonable SLA offers to the end users.

We illustrate this application with an experiment on an SLA addressing execution time. Assuming that participant A receives the request of some user at time $T_{\text{in}}(\texttt{req}) = 0$, we are interested in which guarantees can be offered to the user with respect to $T_{\text{out}}(\texttt{req})$ for a given class of input data. Besides the data, the participant QoS models for A and B depend on several internal activity parameters. t_{send} is the time needed by a participant to deliver the message to a participant mailbox. t_{budget} is the time needed to retrieve budget line information in activity a_3. t_{best} is the time required by activity a_7 to find the best choice among the alternatives offered.

The ranges of values for these parameters are normally empirically established by monitoring. Such empirical data is effectively a sample (or a collection of samples) of the "true population" set from which the QoS metric values are drawn and whose exact bounds are generally unknown. We can use well-known techniques of descriptive statistics on these samples to estimate the parameters of central tendency (mean, median) and dispersion (standard deviation) for the whole population of values. In

that way, we can define intervals whose bounds include the QoS values with some level of confidence. This level will be < 100%, since, in general, total confidence is not attainable. Note that the choice of the confidence level is generally a matter of heuristics. A 99% confidence interval, for instance, is wider (and thus safer) than a 90% one, but, depending on the distribution of values, it may lead to overly conservative predictions and SLA offers to the clients that are safer, but too pessimistic, unattractive, and uncompetitive. The top part of Table 1 lists the ranges of the mentioned component execution time across three experimental confidence levels: 99%, 90% and 80%, with a common range for t_{send}.

The central part of Table 1 shows the ranges for $T_{\text{out}}(\texttt{req})$ obtained by solving the model for each confidence interval in the experiment. In general, for each class of input data sizes, the range of $T_{\text{out}}(\texttt{req})$ contracts, and its maximum, which can be offered as an element of the SLA, decreases when using smaller confidence intervals. To further refine the SLA offer, the provider for participant A can look at the branch condition in a_6, and offer more attractive "fast-track" conditions (with circa 40% reduction in the upper execution time bound) when it becomes known that the `force` flag will be set to `true`, as shown in the lower part of Table 1.

We used the ECL^iPS^e constraint logic programming system [2] which has native support for the integer and real non-linear arithmetic constraints (including disjunctive constraints) that are used in the derivation of the model. Deriving the constraint models with our pilot implementation and solving them with a centralized solver took on average around 260 ms on an i86_64 laptop computer with 4GB of RAM running Mac OS X 10.7.3.

4.2 Predicting SLA Violations at Run Time

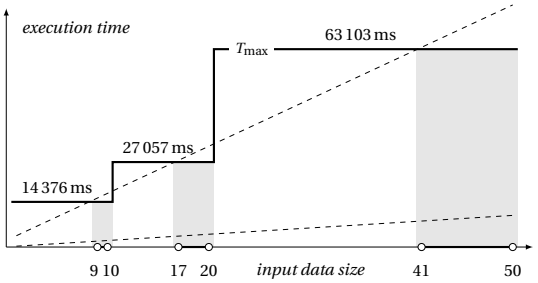

Fig. 8. An example of SLA failure prediction zones.

The constraint-based QoS model can be used for predicting SLA violations at runtime. Since the participant SLA is always related to some event that happens in one of the participants (such as sending the reply in activity a_{11} of our sample choreography), we can apply a variation of the constraint-based prediction method for orchestrations [14]. In that method, we make predictions at each point in execution of the participant processes for which we have the continuation and the monitoring data describing the previously accumulated QoS metrics. In the case of execution time, the imminent failure condition for participant A is predicted when the constraint $T_{\text{out}}(\texttt{req}) \leq T_{\max}$ is proven unsatisfiable in the constraint QoS model, i.e., when SLA compliance cannot possibly be achieved.

Using the experimental settings from the previous subsection, we predict SLA violations for a running choreography with fixed input data size (known at run time), by taking $T_{\max}$ to be the upper bound of $T_{\text{out}}(\texttt{req})$ for the 80% confidence interval in each input data class from Table 1. The thick black line in Figure 8 shows $T_{\max}$

for input data sizes in the range 1..50. The dashed lines show the upper and lower bound of $T_{\text{out}}(\texttt{req})$ for a 99% confidence interval. SLA violations are possible in the gray zones that correspond to data size intervals 9..10, 17..20, and 41..50. In those intervals, imminent SLA violation can be predicted between 175 ms and 325 ms ahead of $T_{\max}$. For other input data sizes (in ranges 1..8, 11..16, and 21..40), the predictor is able to predict SLA conformance at the very start. In both cases, the percentage of correctly predicted cases is typically very high, between 94% and 99% [5].

4.3 SLA Compliance Checking, Dynamic Binding and Adaptation

We now turn to a situation where there exist several implementations for a participant role in a choreography, that are known to be compatible with the communication protocol, message data types, and message cardinalities. We now want to see how the knowledge about participant QoS models can help us rule out some combinations of participant implementations (or promote others) at design time.

For instance, let us take participant A from Figure 1, and assume that there are two implementations that can take the role of B and which differ only in the method for generating alternatives in activity a_{15}: while B_1 can generate one or more alternatives, B_2 always generates at least two. Although the ranges for all participant model variables of B_2 are subsets of the corresponding ranges for B_1, the combination of A with B_2 is illegal for some SLAs and input data sizes for which A with B_1 may work. E.g., for $T_{\max} = 18000$ ms, the constraint model predicts that the combination of A and B_2 is guaranteed to fail for input data sizes of 50 and above when `forced=false` in A. Since A does not control `forced`, for such data sizes it should rule out B_2, and choose B_1 which has a chance to meet the SLA.

This kind of analysis can be performed by checking that every internal structural parameter of A in the constraint QoS model for the choreography (such as the condition in a_6 and the number of iterations of a_4) augmented with condition $T_{\text{out}}(\texttt{req}) > T_{\max}$ has at least one value for which the condition $T_{\text{out}}(\texttt{req}) \le T_{\max}$ is satisfiable for the given range of input data sizes. Alternatively, the same check can be used for dynamic binding at run-time to select an implementation for the role of B for the known size of the particular input request. Such dynamic binding provides a finer-grained per-request selection, at the cost of additional run-time analysis.

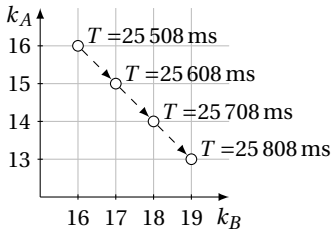

Fig. 9. Adaptation need detection in B.

However, selecting B_1 does not guarantee $T_{\text{out}}(\texttt{req}) \le T_{\max}$: if at run time each invocation of a_{15} happens to return more than one alternative (thus behaving in the same way as B_2), the SLA will be violated for some input data sizes. Participant B can use its QoS model to detect such a situation and to adapt by forcing a_{15} to start returning single items. At the beginning of each iteration in loop a_{14} from Figure 1, B can test whether the execution of a_{15}, if it generates multiple alternatives, can lead to an SLA violation. If so, it can coerce a_{15} to produce a single item and so enforce the SLA. The earliest points in time when that can happen for input data size in range 17..20 and $T_{\max} = 27057$ ms (the central gray zone in Figure 8), are shown in Figure 9.

k_B stands for the previous number of iterations of a_{14}, and k_A stands for the previous number of times when more than one alternative was generated in a_{15}.

5 Conclusions

The constraint-based approach to QoS assurance for service choreographies presented is based on the automatic derivation of QoS constraint models from abstract descriptions of multiple participating processes that can engage in complex, stateful conversations. The QoS attributes that can be modeled include execution time, availability, monetary cost, the quantity of data transferred, and any others that can be mapped onto cumulative, non-negative numerical metrics. For greater precision, the model derivation is augmented with an analysis of message types with size constraints, and the resulting models are data sensitive. The participant models can be derived, integrated, and solved centrally, or in a distributed fashion. The approach can be used at design-time, for classes of input data, and also at run time, with the actual data, whenever the information about the current point in execution is provided for the participants. The resulting models can be used to support SLA negotiation, SLA violation prediction, design-time SLA conformance for classes of input data, dynamic binding of participants, and SLA-driven run-time adaptation.

Based on our prototype implementation, our future work will aim at the development of the supporting tools and systems, and interfacing them with the service infrastructure components, such as the execution engines and service buses, and with choreography design tools. We will also aim at evaluating the quality of QoS prediction offered by the constraint-based models in distributed settings and when used with incomplete or inaccurate information about the QoS properties of the service environment and components.

References

1. K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. Krzysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming Using ECLIPSE.* Cambridge University Press, 2007.
3. Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 191–202, New York, NY, USA, 2012. ACM.
4. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
5. Ivanović D, M. Carro, and M. Hermenegildo. Exploring the impact of inaccuracy and imprecision of qos assumptions on proactive constraint-based QoS prediction for service orchestrations. In *Proceedings of the 4th International Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2012*, pages 931–937. IEEE Press, June 2012.
6. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2nd ed. edition, 2002.
7. Rina Dechter. *Constraint Processing.* Morgan Kauffman Publishers, 2003.
8. Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *ICWS*, pages 296–303, 2007.

9. Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. Sessions and session types: an overview. In *Proceedings of the 6th international conference on Web services and formal methods*, WS-FM'09, pages 1–28, Berlin, Heidelberg, 2010. Springer-Verlag.
10. Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15:313–341, 2008. 10.1007/s10515-008-0032-x.
11. Boi Faltings and Makoto Yokoo, editors. *Artificial Intelligence Journal: Special Issue on Distributed Constraint Satisfaction*, volume 161. Elsevier Science Publishers Ltd., Essex, UK, January 2005.
12. J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In P. Mähönen, K. Pohl, and T. Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2008.
13. D. Ivanović, M. Carro, and M. Hermenegildo. Towards Data-Aware QoS-Driven Adaptation for Service Orchestrations. In *Proceedings of the 2010 IEEE International Conference on Web Services (ICWS 2010), Miami, FL, USA, 5-10 July 2010*, pages 107–114. IEEE, 2010.
14. D. Ivanović, M. Carro, and M. Hermenegildo. Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations. In Gerti Kappel, Hamid Motahari, and Zakaria Maamar, editors, *Service-Oriented Computing – ICSOC 2011*, number 7084 in LNCS, pages 62–76. Springer Verlag, December 2011. Best paper award.
15. Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *ICWS*, pages 369–376. IEEE Computer Society, 2010.
16. A. Metzger, S. Benbernou, M. Carro, M. Driss, G. Kecskemeti, R. Kazhamiakin, K. Krytikos, A. Mocci, E. Di Nitto, and et al. B. Wetzstein. Analytical quality assurance. In *Service Research Challenges and Solutions for the Future Internet*, volume 6500 of *LNCS*, pages 209–270. Springer Verlag, 2010.
17. Object Management Group. *Business Process Modeling Notation (BPMN), Version 1.2*, January 2009.
18. John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation Journal*, 6:233–247, 1993.
19. O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, and K. Pohl. Usage-based online testing for proactive adaptation of service-based applications. In *COMPSAC 2011 – The Computed World: Software Beyond the Digital Society*. IEEE Computer Society, 2011.
20. E. Schmieders and A. Metzger. Preventing performance violations of service compositions using assumption-based run-time verification. In A. Zisman, I. Llorente, Surridge M., Abramowicz W., and Vayssière J., editors, *ServiceWave 2011*, LNCS. Springer, 2011.
21. Sebastian Stein, Terry R. Payne, and Nicholas R. Jennings. Robust execution of service workflows using redundancy and advance reservations. *IEEE T. Services Computing*, 4(2):125–139, 2011.
22. G. Tselentis, J. Dominigue, A. Galis, A. Gavras, and D. Hausheer. *Towards the Future Internet: A European Research Perspective*. IOS Press, Amsterdam, The Netherlands, 2009.
23. Wil M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. M. W. Verbeek. Choreography Conformance Checking: An Approach based on BPEL and Petri Nets. In *The Role of Business Processes in Service Oriented Architectures, Dagstuhl Seminar Proceedings*, 2006.
24. World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0*, November 2005.