Foundations for Web Services Orchestrations: functional and QoS aspects, jointly

Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard

Abstract. Web Services Orchestrations require a firm mathematical basis for their development, regarding both their functional and QoS characteristics. We provide such a basis in the form of a model based on colored Petri net systems. Our approach allows evaluating end-to-end QoS of the orchestration with the help of the QoS of the called sites.

1 Motivation

Web Services (WS) Orchestrations and Choreographies have been the subject of numerous studies and standardisation actions [2,7]. Developing complex orchestrations requires techniques and tools to formally analyse both the functional behavior of an orchestration as well as its Quality of Service (QoS) characteristics. Unfortunately, there are a number of remaining issues regarding WS orchestrations and their formal modeling.

Foundational studies on Web transactions and orchestrations are found, e.g., in [24,8], using abstract state machines, process algebras, or variants of the \(\pi\)-calculus. Several semantic studies have been performed for BPEL, e.g., [4,16]. Studies closest to our are [9,6,15]; they provide a translation of BPEL into Petri Nets of workflow type (without loops back) aiming at property verification.

QoS for WS is an important but delicate issue. It faces the closed/open world paradox: orchestrations are specified as stand alone “closed” entities. Still, they operate in an open environment, by sharing resources with other orchestrations, other Web Services, and other computing and communication activities. In this respect, orchestrations are just another client of a networked infrastructure.

To address the same situation of many users with different QoS sharing a network resource, the following approach is taken by the network community [5]: The global system is considered and a queuing network approach is used. Decentralized mechanisms for control are proposed, in the form of QoS guarantees—typical QoS parameters are end-to-end throughput and jitter. Stability and overall system performance are analysed, globally.

Unfortunately, it is an essential principle that QoS contracts that go together with WS orchestrations, should not refer to and not even know of the existence of the rest of the world. Contracts should make the environment opaque. This is essential because orchestrations would in general traverse different transport domains: resource constraints

⋆ This work is supported by RNRT-SWAN contract from the Agence Nationale de la Recherche. S. Rosario, A. Benveniste, and S. Haar are with IRISA/INRIA, Rennes, C. Jard is with IRISA/ENS Cachan, Rennes.
are attached to these domains but are unknown to the orchestration. Therefore, taking
the above global approach is not relevant for orchestrations. Instead, a contract based
approach is followed, where every different site or agent contributing to the orchestration
exposes its “guaranteed” service model. It is the duty of the orchestration to relate
its sub-contracts to the contract it can offer to its users. Then, sub-contract and contract
monitoring is performed at run-time and drives reconfiguration whenever needed.

In this paper we develop semantic foundations for WS orchestrations, encompassing
both functional and QoS aspects. First, we provide a clean semantic basis for orchestrations
allowing for unbounded but finite recursion, and therefore providing a clean
treatment of dynamic instantiation. To clarify the issue, we have chosen to analyse the
ORC formalism proposed by J. Misra et al. [11] to specify orchestrations. The interest
of ORC is that it is nicely designed, based on few primitive constructs; it implements
the so-called “tree-programming” paradigm, where an initial query can be forwarded
in parallel and/or cascade to other sites that will contribute to building the answer; an-
swers to partial sub-queries are then progressively collected and eventually returned to
the original caller. This paradigm exactly fits the concept of orchestration, it is more
restricted than the model needed to encompass choreographies, where different WS act
as peers. ORC has been used to model different kinds of orchestrations and workflow
patterns [12] and is thus a realistic choice too. Semantic studies for ORC based on se-
quential traces have been developed by Misra et al. [11]. Our semantics is based on
partial orders. It targets Petri net systems [3] with colors, allowing for a finite represen-
tation of infinite nets resulting form dynamic instantiation.

A mild extension of this semantics allows us to capture QoS in a mathematically
sound way by using a probabilistic framework: the formal correlation mechanism that
is provided with token colors allows tracking exceptions, and capturing QoS parameters
is simply performed by adding further colors; QoS composition is obtained by assuming
the called site to be probabilistically independent.

2 Related Work

Many proposals for incorporating quality of service aspects into Web Services have
been made in the past [10,19,22,21]. The common approach is to define and conceptu-
alizing Service Level Agreements (SLA) or to extend existing Web Service description
languages (like WSDL) to specify non-functional attributes. In [22], the authors define
WS-QOS, an XML Schema for specifying QoS properties and provide a framework
for the dynamic selection of Web Services based on QoS attributes which is done with
the help of an intermediate Web Service broker. The Web Service Offerings Language
(WSOL) [27] allows defining different classes of services based on their QoS levels, for
the same functional service.

Though much work has been done on the Web Service composition problem, the ap-
proaches considering QoS aspects while composing Web services are rather limited. In
[23] the authors take a fuzzy logic based approach for the problem. The QoS composi-
tion problem is formalized as a fuzzy constraint which is then solved for. An interesting
work is [26] which considers WS orchestrations with finite execution paths and pro-
poses a Statecharts modeling for these. Composition rules for specific QoS criteria are
proposed that are amenable to optimized configuration of the orchestration with linear programming techniques. Another work with goals similar to ours is [17] which aims to derive QoS parameters for a workflow, given that of its individual tasks. They use a graph reduction technique, repeatedly re-writing the workflow, until only a single task with the deduced QoS attributes of the workflow in consideration, remains.

Our approach differs from all of the above primarily in the fact that we take a probabilistic approach to the problem. Also, we represent the QoS constraints as probabilistic distributions instead of using just hard constraints as commonly is the case.

3 The orchestration model: ORC [11]

3.1 ORC, its syntax and intuitive semantics

The abstract syntax of the ORC and its intuitive semantics are shown in Table 1. ORC expressions specify orchestrations; they return zero, one, or a stream of values. In contrast, site calls return at most one value. Timeouts are special sites that raise time based exceptions. The time that is referenced in timeouts is the only local time that is attached to the orchestration; no time attached to distant sites is required; this avoids classical inconsistency problems regarding time, caused by distribution. Also, in the ORC model, the only mode for a site call is by invocation, service push cannot be captured in ORC. Misra et al. call this restricted programming model “tree programming”. This paradigm does not exhibit all difficulties of full fledged distributed programming and is still adequate for the simpler case of WS orchestrations—but not for choreographies where orchestrations interact as equal peers. We now present our toy example that will support the rest of our presentation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Intuitive Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \in \mathbb{E}$</td>
<td>Expression name</td>
</tr>
<tr>
<td>$S \in \mathbb{S}$</td>
<td>Site</td>
</tr>
<tr>
<td>$x \in \mathbb{V}$</td>
<td>Variable</td>
</tr>
<tr>
<td>$c \in \mathbb{C}$</td>
<td>Constant</td>
</tr>
<tr>
<td>$p \in \mathbb{P}$</td>
<td>Parameter</td>
</tr>
<tr>
<td>$x := F(p)$</td>
<td>Evaluates $F(p)$, assigns 1st value received to $x$</td>
</tr>
<tr>
<td>$F(p)$</td>
<td>Expression call (new instance thereof)</td>
</tr>
<tr>
<td>$S(p)$</td>
<td>Site call, returns at most 1 value or site identifier</td>
</tr>
<tr>
<td>let$(p)$</td>
<td>Publishes the value of $p$</td>
</tr>
<tr>
<td>if$(b)$</td>
<td>Returns a signal if $b$ is true, else does not respond</td>
</tr>
<tr>
<td>$f \mid g$</td>
<td>Symmetric and concurrent parallel composition: the returns from $f$ and $g$ are interleaved</td>
</tr>
<tr>
<td>$f &gt;x&gt; g$</td>
<td>Sequential composition ($x$ optional): each value returned by $f$ causes a fresh evaluation of $g$ and this value can be passed to $g$ via channel $x$</td>
</tr>
<tr>
<td>$f \text{ where } x := g$</td>
<td>Asymmetric parallel composition: the 1st value produced by $g$ is passed to $f$ via channel $x$</td>
</tr>
</tbody>
</table>

Table 1. The abstract syntax of ORC and its intuitive semantics
4 The CarOnLine illustrative example

The example is shown in Table 2. The service described consists in getting the pair

\[
\text{Client} : \text{in let}(\text{BestCarPrice}, \text{BestCredit})
\]

where \( (\text{BestCarPrice}, \text{BestCredit}) : \text{in CarOnLine} \)

\[
\text{CarOnLine} =
\quad \text{CarPrice} > \text{BestCarPrice} >
\quad \{ \begin{array}{l}
\quad \{ \text{if}(\text{BestCarPrice} \neq \text{Fault}) \} > \text{CreditRate}(\text{BestCarPrice}) > \text{BestCredit} >
\quad \text{let}(\text{BestCarPrice}, \text{BestCredit}) \\
\quad \{ \text{if}(\text{BestCarPrice} = \text{Fault}) \} > \text{CarOnLine} \\
\end{array} \}
\]

\[
\text{CarPrice} =
\quad \text{broadcast}(\text{GarageList}) > \text{values} > \text{Min}(\text{values})
\]

\[
\text{Min}(\text{values}) \text{ returns } \text{Fault} \text{ if } \text{values} = \text{Fault} \text{ and otherwise it returns the minimum among the tuple of received (valid) values; broadcast is defined as follows:}
\]

\[
\text{broadcast}([[]]) = \text{let}(\text{Fault})
\]

\[
\text{broadcast}(g:g) = \text{mux}(u, v) \text{ where}
\]

\[
\begin{array}{l}
\quad \text{u} : \text{in} \{ g | \text{Rtimer}(\text{timeout}) > \text{let}(\text{Fault}) \}
\quad \text{v} : \text{in} \text{broadcast}(g)
\end{array}
\]

\[
\text{mux}, \text{the “multiplexer”, is a site call used to filter out the “Fault” values due to the timeout; for all valid values } x \text{ and } y, \text{mux is defined as:}
\]

\[
\begin{array}{l}
\quad \text{mux}(x, y) = (x, y)
\quad \text{mux}(x, \text{Fault}) = x
\quad \text{mux}(\text{Fault}, y) = y
\quad \text{mux}(\text{Fault}, \text{Fault}) = \text{Fault}
\end{array}
\]

Table 2. The CarOnLine ORC program.

\( (\text{BestCarPrice}, \text{BestCredit}) \) from the CarOnLine service. CarOnLine decomposes into the following sequence of operations: 1/ getting, from CarPrice, the BestCarPrice; 2/ the latter is passed as a parameter to CreditRate service, which returns BestCredit; if CarPrice returns an exception “Fault”, then the query is reemitted (recursive call). Service CarPrice is a broadcast of the same query to a pool of garages. Each garage of the list may return a price or an exception “Fault”, emitted on time out—note that the Rtimer sits on the orchestration site, so that no reference to global time is made. Observe that exceptions are described as part of the orchestration itself; this is the normal way of dealing with exceptions when specifying WS orchestrations.
5 From ORC to colored Petri net systems

In this section, we formally define the semantics of ORC by a finite representation in terms of systems of colored Petri net equations. This is a net-based formalism proposed by [3] that allows describing unbounded nets in a finite manner; such unbounded nets may arise due to recursion. In this paper we only give the principles of the translation. A detailed description can be found in [18].

5.1 Reflecting the ORC programming model

Each ORC expression is structurally translated into a colored Petri net with the features shown in Fig. 1 and discussed next.

Fig. 1. Generic form for the Petri net translation of an ORC expression. The place labelled \( a \) is the activation place. The places labelled \( x_1, \ldots, x_n \) are the parameter places of the ORC expression, i.e., the expression is of the form \( f(x_1, \ldots, x_n) \). The double-bordered place in red is the power place of the net. The place labelled \( r \) is the return place of the net. The non directed arcs denote read arcs. The doubly directed arc from the power place to \( f \) represents a set of arcs from the power place to transitions in \( f \) and from transitions in \( f \) to the power place—this set of arcs varies for each different expression.

Each ORC expression possesses a single activation point and returns a single (possibly empty) stream of values. Therefore, its Petri net translation has a special minimal place that we call activation place and a distinguished return place for storing the returned values. The start of the execution of an expression corresponds to placing a token in the activation place. An ORC expression may use parameters for its execution; The Petri net of an ORC expression possesses one minimal place for each of these parameters; these places are called parameter places, they are labeled by their corresponding parameter name. Finally, a special place, called the power place, is introduced to properly control the termination of the expression—killing an expression is performed by “shutting power down”.

The translation targets colored Petri nets with the following kind of arcs: Ordinary directed arcs, consuming and producing tokens. Read arcs, requiring the presence of tokens in their source node for their sink transition to fire; read arcs do not consume tokens—these arcs are needed to model the fact a variable bound to a value may be referenced for an unknown (possibly infinite) number of times; read arcs are used in
particular for parameter places. Reset arcs, which remove all tokens from their anterior place, disabling the subsequent firings of the posterior transitions.

5.2 The Coloring mechanism

In our translation, each expression $E$ (and so even the sub-expressions of $E$) has a unique net corresponding to it in this system of nets. The net for an expression $E$ will be denoted by $N_E$ from now on.

Colors. Colors for tokens in our translation serve many purposes: they are used to distinguish different activations of the same expression, to match the control and data tokens in a site call, and to specify termination of expressions. Colors are tuples of attributes; by abuse in the sequel, we simply call “color” any of these attributes. We adopt the following conventions for the coloring of tokens:

- A single ORC program may be activated more than once. These different activations are distinguished by a distinct color.
- By interleaving the results returned by each branch, the symmetric parallel composition operator “|” of ORC enables the creation of a stream of values from a single activation, see Table 1. We will need to distinguish the different tokens of such streams and so we append distinct colors to each different control token created in a parallel construct.
- There could be more than one instance of the same expression $E$, while there is only one net $N_E$ corresponding to its definition. Thus, each instance is given a unique label, called its instance label.

To distinguish the different activations of the same expression we append the instance label of the calling instance to the instance color of each token while transferring it, from the activation place (and parameter places) of the instance of $E$, to the activation place (and parameter places) of $N_E$. Thus the instance color is a list. The instance label is removed from the token instance color when the call returns.
- Transitions corresponding to site calls add a data color to the token, which is the data value returned by the site call.

As a result, we define the color of each token to be a tuple $(\text{Id}, \text{pList}, \text{hList}, \text{Data})$:

- $\text{Id}$ is the identifier color, i.e., the distinct color added at the start of each different activation;
- $\text{pList}$ is the list of colors added by the parallel constructs; initially this list is empty, successive parallel constructs append distinct colors to this list;
- $\text{hList}$ is the instance color, which is a list of labels added during expression calls;
- $\text{Data}$ is list of data colors.

The first three components $(\text{Id}, \text{pList}, \text{hList})$ correspond to the identifier part of the token. Throughout this section,

$$c = (i, p, h, v) = (\text{Id}, \text{pList}, \text{hList}, \text{Data})$$

shall denote a generic token color as above. For $c$ as above, we shall denote by $i_c$ the $i$-color of $c$ and so on. For a color $c'$, write simply $(i', p', h', v')$ to denote its components.
Finally, if \( c \) is indexed, e.g., \( c = c_1, c_2, \ldots \), we write \( i_1, i_2, \ldots \), for the corresponding \( i \)-color.

The matching relation. Colors will play a central role in controlling the firing of transitions. More precisely, to each transition \( t \) of net \( N_E \) we shall attach a partial function \( F_t \), mapping a tuple of input colors to a tuple of output colors. We call \( F_t \) the firing rule of \( T \).

The domain of each firing rule, i.e., the set of allowed input color configurations, will be expressed in terms of a special family of constraints involving the following matching relation defined over pairs of colors:

\[
e \sqsubseteq e' \iff (i = i') \land (h = h') \land (p \in \text{prefix}(p'))
\]

where \( \text{prefix}(p') \) is the set of all prefixes of \( p' \). In our comprehensive translation of ORC [18], the different firing rules are specified for each ORC primitive. In this paper, we shall indicate them when analysing our illustration example in Section 6.

5.3 The marking equivalence

Each ORC program has a main expression, which is first called when starting the execution of an orchestration. Each activation of an ORC program proceeds first by placing tokens in the activation place of the net for the main expression. Then, it proceeds by calling the expressions (and sub-expressions) which constitutes it. The calling of an expression \( E \) transfers a token from the activation place of the instance of \( E \) to the activation place of \( N_E \). The modeling of this calling (and return) of expressions is done by using a marking equivalence relation between an instance of \( E \) and its corresponding net \( N_E \) [3].

Let \( I_E \) denote an instance of \( E \) with an instance label \( l \), and let \( N_E \) be the net corresponding to \( E \). Let \( a_{I_E} \) and \( a_{N_E} \) denote the activation places of \( I_E \) and \( N_E \) respectively, and let \( PW \) denote the power place. Let \( c = (i, p, h, v) \) be a token of the form described in section 5.2 and let \( c.P \) denote the marking with token \( c \) in the place \( P \) and nothing elsewhere. Then for a marking \( M \), the calling of expression \( E \) from an instance \( I_E \) is given by the following relation:

\[
M + c.a_{I_E} + c.PW \equiv M + (i, p, h, l, v).a_{N_E} + (i, p, h, l, v).PW
\]

Essentially, the instance label \( l \) of \( I_E \) is added to the hList component of the token before transferring it to the activation place of \( N_E \).

The return of an expression call moves a token from the return place of \( N_E \) to the return place of the instance from which it was called. The instance label added during the expression call is removed from the token color while transferring it back. If \( r_{I_E} \) and \( r_{N_E} \) denote the return places of \( I_E \) and \( N_E \) respectively, the equivalence relation for the return of an expression call is given by:

\[
M + (i, p, h, l, v).r_{N_E} + (i, p, h, l, v).PW \equiv M + c.r_{I_E} + c.PW
\]

where \( l \) is the instance label of \( I_E \).
Marking equivalence relations also exist between the parameter places of $I_E$ and $N_E$. They are handled in a similar way. The only ORC construct that will not contain instances as its sub-expressions are the most basic expression i.e., site calls. As a result, the execution of a site call is completely defined by standard Petri net firing rules, without any marking equivalence relation.

5.4 Translating Site Calls

We illustrate our principles for the translation of site calls, which are the most basic ORC expressions. Site calls may use a list of parameters, all of whose values will have to be defined before the site call can happen.

![Site call diagram]

Fig. 2. Site call $S(x_1, x_2, \ldots, x_n)$ and its firing rule.

The Petri net corresponding to a generic site call is shown in Figure 2. The labels on the arcs indicate the color of the tokens that pass through them. The firing rule of transition $S$ is given. It consists of $n$ constraints $c_i \subseteq c$, $i = 1, \ldots, n$, together with the output function $c' = (i, p, h, S(v_1, \ldots, v_n))$ (recall that $v_i$ denotes the $v$-component of color $c_i$).

Every activation of a site call occurs by placing controls tokens in the activation place $a$. The call will proceed only if the expression which called the site is “active” (i.e., has not been terminated) which is ensured by the arc from the power place to the transition following $a$. The value component of the return token is set to the value $S(v_1, \ldots, v_n)$ returned by the site call as shown in the figure. The first three components of the token added to the return place are the same as that of the initial control token activating the site call. A copy of the new token is stored in the power place too as shown.

6 Translating the CarOnLine example

Figures 3, 4, and 5, show the PN translations of the three components CarOnLine, CarPrice, and Broadcast, respectively. We show the evolution of the colors of a given token when it traverses the different places of the net.
Fig. 3. CarOnline and its firing rules. Note the recursive call of CarOnline.

General comments and conventions: The following notational conventions are used to describe the firing rule of each transition. Color $c_1$ decomposes as $c_1 = (i_1, p_1, h_1, v_1)$ and similarly for other color names. A distinct label is attached to each transition, e.g., $T_1, T_2, \ldots$ Transitions denoting site calls are white; other transitions, added for the purpose of the translation, are black.

For each transition, we give the firing rules as a set of constraints on input colors (e.g., $v_1 = []$ for $T_1$ and $T_2$ in Figure 5) and the equations giving the resulting set of output colors. In these equations, names of colors are local to each transition and are specified in the diagrams; for example, referring to Fig. 5, transition $T_3$ has $c$ and $c_1$ as input colors and $c'$ as output color. Since names are local to transitions, they are reused across the diagram without referring necessarily to identical colors.

For each transition, the activation token is systematically denoted by $c$; the constraint $c' \subseteq c$ always holds, where $c$ is the color of activation token and $c'$ is the color of any input token; to simplify the diagrams, these systematic constraints are not mentioned. The power place and its related arcs are omitted from Figures 3, 4, and 5.

Figure 3 A simplification has been performed in this figure. The arc leaving the transition BestCarPrice, which is shown as branching into four arcs, actually represents four different arcs with the same arc label $(i, p, h, bcp)$. The parallel appends a label $t_i, i = 1$ or 2 to the pList, for its two branches.
Figure 4. CarPrice and its firing rules.

Figure 5 This figure exhibits a “where” expression with its two “:∈” subexpressions located at transitions $T_{10}$ and $T_{11}$, which extract only the first token in case a stream of tokens is generated—this may occur when an expression is called in the scope of the where; e.g., here Broadcast is re-called. This is achieved by matching token colors: according to (2), two tokens can synchronize at the “v :∈” transition iff they possess identical hList color, i.e., $h' = []$; this constraint selects only the 1st token produced.

7 QoS of ORC orchestrations

In this section, we investigate how to enhance our Petri net system with further attributes and colors to encompass QoS analysis. As explained in Section 1 we follow a contract based approach, where we abstract through a SLA the interaction with the external world that results from other users calling some sites used in the orchestration.

7.1 SLA modeling and analysis

Contracts and SLA can be seen as an opaque interface between the considered orchestration and the external world. SLA are promises offered, by the called sites, to the orchestration they contribute to. SLA parameters can be numerous. They typically combine the following three types of information: 1/ the allowed throughput of queries under which other promises are offered; 2/ the availability (encoded by the type of exception); and 3/ the response time for a valid response.

SLA modeling The mathematical modeling of contracts is a difficult issue. Contracts typically have a statistical nature: “the service shall answer properly for 95% of the cases”, and, when answering properly, “it shall answer in 90% of the cases with less than 200msec”.

\[ T : [ \lambda' = (i, p, h, \text{min}(v_1))] \]
To handle this statistical aspect, we propose a probabilistic approach—hard constraints can be handled too, just as particular cases. We formalize the SLA of a site by means of a probability distribution $P_{\text{contract}}$ that the considered site exposes to its environment. The distribution $P_{\text{contract}}$ specifies how the site behaves.

Regarding SLA parameters, throughput is by itself an “averaged” concept and is not the right candidate for a random variable. Instead, we consider the random delay $\delta$ between two successive queries. With this modification, probability $P_{\text{contract}}$ governs the triple

$$(\delta, \varepsilon, \tau) = (\text{inter-query time, exception, response time})$$

that characterizes it. Note that these three components play a non-symmetric role: the inter-query time characterizes the stimulation of the site by the orchestration (the site is

\[ T_1 : \left[ v_1 = [ ] \right] \quad T_2 : \left[ v_1 = [ ], \ c' = c \right] \quad T_3 : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_4 : \left[ c_1 = (i, p.t_1, h, v) \right] \quad T_5 : \left[ c_1 = (i, p.t_4, h, v) \right] \quad T_6 : \left[ c' = (i, p, h, call(v_1)) \right] \]

\[ T_7 : \left[ c' = (i, p, h, v_1) \right] \quad T_8 : \left[ c' = (i, p, h, v_1) \right] \quad T_9 : \left[ c' = (i, p, h, \text{Mux}(v_1, v_2)) \right] \]

\[ T_{10} : \left[ c = (i, p.t_2, h, v) \right] \quad T_{11} : \left[ c = (i, p.t_3, h, v) \right] \quad T_{12} : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_{13} : \left[ v_1 = [ ] \right] \quad T_{14} : \left[ v_1 = [ ], \ c' = c \right] \quad T_{15} : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_{16} : \left[ c_1 = (i, p.t_1, h, v) \right] \quad T_{17} : \left[ c_1 = (i, p.t_4, h, v) \right] \quad T_{18} : \left[ c' = (i, p, h, call(v_1)) \right] \]

\[ T_{19} : \left[ c' = (i, p, h, v_1) \right] \quad T_{20} : \left[ c' = (i, p, h, \text{Mux}(v_1, v_2)) \right] \]

\[ T_{21} : \left[ c = (i, p.t_2, h, v) \right] \quad T_{22} : \left[ c = (i, p.t_3, h, v) \right] \quad T_{23} : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_{24} : \left[ v_1 = [ ] \right] \quad T_{25} : \left[ v_1 = [ ], \ c' = c \right] \quad T_{26} : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_{27} : \left[ c_1 = (i, p.t_1, h, v) \right] \quad T_{28} : \left[ c_1 = (i, p.t_4, h, v) \right] \quad T_{29} : \left[ c' = (i, p, h, call(v_1)) \right] \]

\[ T_{30} : \left[ c' = (i, p, h, v_1) \right] \quad T_{31} : \left[ c' = (i, p, h, \text{Mux}(v_1, v_2)) \right] \]

\[ T_{32} : \left[ c = (i, p.t_2, h, v) \right] \quad T_{33} : \left[ c = (i, p.t_3, h, v) \right] \quad T_{34} : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_{35} : \left[ v_1 = [ ] \right] \quad T_{36} : \left[ v_1 = [ ], \ c' = c \right] \quad T_{37} : \left[ c' = (i, p, h, v_1) \right] \]

\[ T_{38} : \left[ c_1 = (i, p.t_1, h, v) \right] \quad T_{39} : \left[ c_1 = (i, p.t_4, h, v) \right] \quad T_{40} : \left[ c' = (i, p, h, call(v_1)) \right] \]

\[ T_{41} : \left[ c' = (i, p, h, v_1) \right] \quad T_{42} : \left[ c' = (i, p, h, \text{Mux}(v_1, v_2)) \right] \]

\[ T_{43} : \left[ c = (i, p.t_2, h, v) \right] \quad T_{44} : \left[ c = (i, p.t_3, h, v) \right] \quad T_{45} : \left[ c' = (i, p, h, v_1) \right] \]
not responsible for it), whereas exception and response time are under responsibility of the site.

For example, exposing $\mathbb{P}_{\text{contract}}$ implies that the following kind of guarantee can be properly stated as part of the SLA (the distinguished value $\top$ for $\epsilon$ indicates that the response was valid, i.e., no exception occurred):

$$\mathbb{P}_{\text{contract}}(\epsilon \neq \top \mid \delta \geq 20) \leq 5\%$$  \hfill (3)

(response is invalid for less than 5% of the queries, given that the inter-query time is not less than 20 msecond), or

$$\mathbb{P}_{\text{contract}}(\tau \leq 5\text{ms} \mid \delta \geq 20, \epsilon = \top) \geq 90\%$$  \hfill (4)

(given that response is valid and the inter-query time is not less than 20 mseconds, then the response is received within at most 5ms for more than 90% of the cases).

**SLA composition** For $O$ an orchestration calling sites $(S_i)_{i \in I}$, we consider the sites $S_i$ as probabilistically independent when composing the orchestration:

$$\mathbb{P}_{O\text{contract}} = \prod_{i \in I} \mathbb{P}_{S_i\text{contract}}$$  \hfill (5)

This means that the triples $(\delta_i, \epsilon_i, \tau_i)$ are independent random variables for different $S_i$. How the corresponding random SLA parameters are effectively composed is formally described in Section 7.2, where we explain how the colors of tokens can be enriched with SLA parameters.

**SLA monitoring** SLA monitoring consists of the following two issues: 1/ the orchestration wants to monitor that each called site responds according to its promises, and 2/ each called site has to monitor that the orchestration is not over-sollicitating it.

From the point of view of the called site, this consists in testing that the random inter-query period (which is the inverse of the throughput) is not stochastically larger than the agreed one.

From the point of view of the orchestration, this consists in testing the following: given that throughput conforms the SLA, then 1/ availability is at least the one promised, and 2/ restricted to the subset of correct answers, the random response time is not stochastically larger than promised.

For $X$ and $Y$ two random variables with values in some totally ordered space $\mathcal{X}$, following [14], say that $X$ is not stochastically larger than $Y$, written

$$X \preceq Y \iff \forall x \in \mathcal{X}, \mathbb{P}(X \leq x) \geq \mathbb{P}(Y \leq x)$$  \hfill (6)

If $B$ is a random predicate, write

$$X|B \preceq Y|B \text{ if } \forall x \in \mathcal{X}, \mathbb{P}(X \leq x \mid B) \geq \mathbb{P}(Y \leq x \mid B)$$  \hfill (7)

where $\mathbb{P}(X \leq x \mid B)$ denotes the conditional probability of $X \leq x$ given that $B$ holds.
In the following, subscripts \( \text{cont} \) and \( \text{act} \) refer to the contracted and actual QoS parameters, respectively; \( \delta \) is the inverse of the throughput, i.e., the random inter-query time, \( \varepsilon \) is the exception, and \( \tau \) is the response time. With these definitions, QoS compliance is defined by the following set of conditions:

\[
\delta_{\text{act}} \geq \delta_{\text{cont}} \tag{8}
\]

\[
P(\varepsilon_{\text{act}} \neq \top | \delta_{\text{act}} \geq \delta_{\text{cont}}) \leq P(\varepsilon_{\text{cont}} \neq \top | \delta_{\text{act}} \geq \delta_{\text{cont}}) \tag{9}
\]

\[
\tau_{\text{act}} | (\varepsilon_{\text{act}} = \top, \delta_{\text{act}} \geq \delta_{\text{cont}}) \preceq \tau_{\text{cont}} | (\varepsilon_{\text{act}} = \top, \delta_{\text{act}} \geq \delta_{\text{cont}}) \tag{10}
\]

Condition (8) defines the monitoring, by the called site, of the orchestration inter-query time. Condition (9) defines the monitoring, by the orchestration, of the availability of a given called site. Finally, condition (10) defines the monitoring, by the orchestration, of the response time of the same called site, given that the response of this site is valid—symbol ‘\( \preceq \)’ in the predicate denotes conjunction.

In these three conditions, contract QoS parameters are agreed probability distributions. In contrast, actual QoS parameters are observed empirical distributions of the corresponding quantities. Therefore, conditions (8–10) consist in comparing, for the stochastic order \( \preceq \), empirical probability densities to agreed probabilities. Statistical tests can be used for this purpose [1].

### 7.2 Enhancing PN systems with QoS attributes

According to the previous section, we need to equip the tokens with additional colors to capture QoS features.

**Dates** A date is attached to each token. This date indicates when the considered token entered the considered place. In particular, the entrance date of the main expression by the token is the date where the corresponding query was issued to the orchestration. Note that the dates considered uses the time that is local to the orchestration, and they can thus be observed by the orchestration. This date is added to the colors of the token as an additional color \( d \).

**Exceptions** Exception attributes are attached to tokens as follows. Exceptions are explicitly specified as part of the orchestration. In the CarOnLine example, an exception is raised upon timeout in the CarPrice expression. The exception returns Fault as a response to CarPrice. In ORC, exceptions are just like any other value returned by expressions. For QoS purposes, however, we need to keep track of exceptions. This is achieved by creating a new (possibly empty) color list, which we call the exception color and denote by \( \varepsilon \).

**Inter-query time** An inter-query time \( \delta \) is attached to the PN system, globally. It represents the random time interval separating two successive queries entering the net. Note that there is no need for an additional attribute for this, since \( \delta \) is just the difference between the entry dates of two successive tokens.
7.3 SLA prediction

SLA prediction is useful for the dimensioning of contracts, by the orchestration. It consists in predicting, from the QoS contracts agreed with called sites, what contract can be offered to the customer, by the orchestration. With contracts of one of the forms (3) or (4), this is a difficult tasks. Simple heuristic rules for combining SLA parameters directly would typically result in significant overshoot and waste of resources. Instead, we simply propose a Monte-Carlo approach to this problem.

To this end, multiple runs of the QoS-enhanced net system $N_O$ associated with an orchestration $O$ are drawn at random. In drawing the triples $(\delta, \varepsilon, \tau)$ for each site $S$, the latter are equipped with the contract probability distribution $P_S$ they promise. When these parameters are drawn, all QoS colors follow. The probability distribution $P_O$, which draws the end-to-end QoS characteristics of the considered ORC program, is then approximated by its empirical estimate and contract parameters for the orchestration follow.

7.4 The Broadcast example

We detail the firing rules of the Broadcast example enhanced with dates and exceptions. The dates and exceptions are shown in red—see Fig. 5 for a diagram of Broadcast. The generic form for a color is now $c = (i, p, h, d, \varepsilon)$, where $d$ is the date of the token and $\varepsilon$ is its exception label ($\varepsilon$ is omitted when empty).

Regarding dates and response times, transition $T_0$, corresponding to the call of some garage from the pool, takes some time for its response. This is modeled by the output equation on dates: $d' = \max(d, d_1) + \tau_{\text{garage}}$. For other transitions, the firing just requires the availability of matching tokens; resulting date for the output is therefore the max in input dates of matching tokens. Note that we assume in this example that local operations of the orchestration take zero time. Such local operations include creating and handling the various threads and performing data manipulations. This is a simplifying assumption that can be refined.

$$
T_1: \begin{cases}
\delta = [] \\
\varepsilon' = \varepsilon \cup \varepsilon_1
\end{cases}
T_3: \begin{cases}
\varepsilon' = (i, p, h, v_1, d, \varepsilon) \\
\delta' = \max(d, d_1)
\end{cases}
T_2: \begin{cases}
\delta = [] \\
\varepsilon_1 \\
\varepsilon' = (i, p, h, v, d', \varepsilon') \\
c_2 = (i, p, h_1, \text{head}(v_1), d', \varepsilon') \\
\delta' = \max(d, d_1) + \tau_{\text{garage}}
\end{cases}

T_4: \begin{cases}
c_1 = (i, p, t_2, h, v, d, \varepsilon) \\
\delta_1 = (i, p, t_3, h, v, d, \varepsilon) \\
\varepsilon_1 \\
\varepsilon' = \delta_1 \cup \varepsilon_1
\end{cases}
T_5: \begin{cases}
\varepsilon' = (i, p, h, v_1, d', \varepsilon') \\
\delta' = \max(d, d_1)
\end{cases}
T_6: \begin{cases}
\varepsilon' = (i, p, h, \text{call}(v_1), d', \varepsilon') \\
\delta' = \varepsilon \cup \varepsilon_1 \cup \varepsilon_{\text{garage}}
\end{cases}

T_7: \begin{cases}
\varepsilon' = (i, p, h, v_1, d', \varepsilon') \\
\delta' = \max(d, d_1)
\end{cases}
T_8: \begin{cases}
\varepsilon' = (i, p, h, v_1, d', \varepsilon') \\
\delta' = \max(d, d_1)
\end{cases}
T_9: \begin{cases}
\varepsilon' = (i, p, h, \text{Max}(v_1, v_2), d', \varepsilon') \\
\delta' = \max(d, d_1, d_2)
\end{cases}

T_{10}: \begin{cases}
\varepsilon' = (i, p, t_2, h, v) \\
\delta' = \max(d, d_1)
\end{cases}
T_{11}: \begin{cases}
\varepsilon' = (i, p, t_3, h, v) \\
\delta' = \max(d, d_1)
\end{cases}

This timing model obeys the so-called MaxPlus algebra [5] in that site or expression
calls induce a delay (which is added to the input dates) whereas synchronizing transitions output the maximum of the dates of incoming tokens.

Regarding exceptions, these can be created either when a timeout occurs (transition $T_8$), or if a garage returns an invalid response (transition $T_6$). They are appended to the exception list.

End-to-end QoS characteristics for the orchestration are found by comparing QoS colors, for tuples of input and output matching tokens.

8 Conclusion and future work

We have developed a comprehensive fundamental framework for Web Services Orchestration, encompassing both functional and QoS aspects. This framework relies on an abstract model in the form of colored Petri net systems and encompasses recursion. To make the presentation of things cleaner, we have chosen to build on top of the ORC formalism; however, the same principles would apply to BPEL.

We have developed a modeling tool in Java that supports this approach. The tools takes as inputs the orchestration described in ORC and the QoS distributions of the sites involved in the orchestration, and performs Monte-Carlo simulations for the orchestration’s QoS dimensioning. The empirical end-to-end QoS attributes of the orchestration can be visualized and orchestration SLA contracts can be derived from this. In the future, we plan to address contract monitoring via statistical techniques. We mention that the group of Misra et al. has developed an ORC tool [13], for the functional aspects only.

References