QoS-Aware Management
of Monotonic Service Orchestrations

A 5 years project, jointly developed with A. Benveniste and 2 PhDs at IRISA/INRIA, in collaboration with Misra’s group in Austin UT

June 22, 2012
Introduction

Wide-area computing

- Services are building blocks for creating open distributed applications
- Services may be composed together to form new services (orchestrations, choreographies)
- Importance of contracts in an open world (SLAs), including non-functional aspects (latency, security, cost, ...)
- Managing business processes over a Web infrastructure
- The example of ORC programming language (J. Misra, Austin), as an clean alternative of BPEL
Typical example

- A typical example alike travel services: a service is composed by reusing existing services exposed by other providers seen as sub-contractors.
- Garantees must be offered:
  - Functional: the composed service shall offer what it is supposed to
  - QoS: with some agreed security and performance (SLA)
Small example in a Petri net style

SubmitOrder(location, budget)

best(cost)

Company\textsubscript{1} \quad Company\textsubscript{2}

Hotel\textsubscript{A} \quad Hotel\textsubscript{B}

best(latency) \quad enough budget?

Timeout \quad Invoice \quad Resubmit(budget)

- Data-dependent workflow
- Multi-dimensional QoS
Small example in a Petri net style

```
SubmitOrder(location, budget)
```

```
best(cost)
```

```
Company_1
```

```
best(cost, cat)
```

```
Hotel_A
```

```
best(latency)
```

```
Hotel_B
```

```
enough budget?
```

```
Timeout Invoice Resubmit(budget)
```

- Data-dependent workflow
- Multi-dimensional QoS
QoS analysis (quite different from networks)

- Combining transactional Web services
  - Seen as “black-” or “grey-boxes”, exposed through their semantically rich interface (WSDL++, WSLA++, ...)
  - Infrastructure-agnostic (SOAP, REST)
- Semi-open world
  - Typically professional
  - Extranet, E-enterprise, E-business
  - Business management
  - Good balance btw functionaionality, security, safety/correctness, and QoS
- Tangency with automation management, and, to a lesser extent, manufacturing systems design
- a world of contracts
Outline

Introduction

Monotonicity in QoS

QoS computation

Implementation in ORC

Soft contracts

Monitoring

Conclusion
Monotonicity

Implicit assumption in contract-based management:
QoS improvements in component services can only be better for the composite service.

- Can be false...
Non monotonicity

SubmitOrder(location, budget)

best(cost)

Company\textsubscript{1} 10\hspace{1cm} 20\hspace{1cm} Company\textsubscript{2}

Hotel\textsubscript{A} 10\hspace{1cm} 20\hspace{1cm} Hotel\textsubscript{B}

best(latency)

enough budget?

Timeout Invoice

Resubmit(budget)

End-to-end cost $= 20$
Cost of \textit{Company}_2 has been improved to 5

End-to-end cost = 25 is worse!
Theorems

- Loose monotonicity: considering maximum QoS for all possible branching choices ensures monotonicity. May lead in practice to very pessimistic QoS estimations.
- Computing branching cells (by unfolding) allows for detection of non monotonicity. Monotonicity is undecidable in general.
- A syntactical sufficient condition for monotonicity is that, each time branching has occurred in net $N$, a join occurs right after.
QoS domain

- Partially ordered domain \( Q = (D, \leq, \oplus, \triangleleft) \) that is a complete upper lattice (the least upper bound operator \( \lor \) meaning taking the “worst” QoS and is used during synchronization)

- Operator \( \ominus : D \times D \to D \) captures how a transition increments the QoS value. \( \ominus \) must be monotonic w.r.t. \( \leq \)

- Competition function \( \triangleright : D \times D^* \to D \) (must be also monotonic)
Examples of QoS domains

- Latency: $Q = (R^+, \leq, +, \triangleleft)$ where $d_1 \triangleleft d_2 = d_1$ (the winner is the first arrived)
- Security: $Q = (\{low, high\}, high \leq low, \lor, \triangleleft)$
- Cost: $Q = (\{1\} \rightarrow N, \subseteq, +, \triangleleft)$
- Composite QoS (product): $Q = ((D_1, D_1), \leq_1 \times \leq_2, +, (\triangleleft_1, \triangleleft_2))$
- Composite QoS (priority): suppose $Q_1$ is security and $Q_2$ is latency.
  - $\leq$ is the lexicographic order from $(\leq_1, \leq_2)$
  - $(s, d) \triangleleft (s', d') = \text{ if } d \leq d' \text{ and } s = low \text{ then } (s, d') \text{ else } (s, d)$ (wait is needed to decide who wins the competition)
QoS computation

Tokens bring the QoS information

- If \(((q_1 \lor q_2) \oplus \delta_t) \leq (q_2 \oplus \delta_{t'})\) then \(t\) fires and \(q' = ((q_1 \lor q_2) \oplus \delta_t) \triangleleft (q_2 \oplus \delta_{t'})\)
- If \(((q_1 \lor q_2) \oplus \delta_t) \geq (q_2 \oplus \delta_{t'})\) then \(t'\) fires and \(q'' = (q_2 \oplus \delta_{t'}) \triangleleft ((q_1 \lor q_2) \oplus \delta_t)\)
- Else choose non deterministically to fire \(t\) or \(t'\)
**Sites:** the fundamental unit of computation. Similar to functions but may be remote and therefore unreliable. Publishes the value returned by the site.

**Combinators:** only four:
- do $f$ and $g$ in parallel: $f \parallel g$
- for all $x$ from $f$ do $g$ (sequential composition): $f > x > g$
- for some $x$ from $g$ do $f$ (pruning): $f < x < g$
- if $f$ completes without publishing do $g$ (otherwise): $f ; g$

- functions
- a lot of built-in sites
Symmetric composition \( f \mid g \)

- Evaluate \( f \) and \( g \) independently
- Publish all values from both
- No direct communication of interaction between \( f \) and \( g \). They can communicate only through sites.

**Example:**

\[ CNN(d) \mid BBC(d) \]

returns 0, 1 or 2 values.
Implementation in ORC

Sequential composition $f >x> g$

- For all values published by $f$ do $g$
- Publish only the values from $g$

**Example:**

\[
CNN(d) >x> Email(address, x)
\]

**Example:**

\[
(CNN(d) \mid BBC(d)) >x> Email(address, x)
\]

may call Email twice.
Pruning $f \prec x \prec g$

- Evaluate $f$ and $g$ in parallel. Site calls that need $x$ are suspended.
  
- **Example:**
  
  $$(M() \mid N(x)) \prec x \prec g$$

- When $g$ returns a (first) value, bind the value to $x$, terminate $g$ and resume suspended calls.

- **Example:**

  $$Email(address, x) \prec x \prec (CNN(d) \mid BBC(d))$$

  sends at most one email.
Fork-join parallelism

- Call $M$ and $N$ in parallel
- Return their values as a tuple after both respond
- **Example:**
  \[
  ((u, v) < u < M()) < v < N()
  \]
Otherwise \( f \; ; \; g \)

Do \( f \). If \( f \) completes without publishing then do \( g \).

- An expression completes if its execution can take no more steps, and all called sites have either responded, or will never respond.
- All library sites in ORC are helpful (indicate if they halt).
- **Example:**

\[
(h \triangleright x \triangleright println(x) \triangleright if\text{t}(false)) ; "done"
\]

- **Example:** print all publications of \( h \). When \( h \) completes, publish “done”.

Concurrent function calls

```
def Metronome() = signal | (Rwait(1000) >> Metronome())
    (Metronome() >> "tick") | (Rwait(500) >> Metronome() >> "tock")
```
Causality and QoS

Goal:
- Specified as an ORC program transformation: $P \rightarrow P'$
- $P'$ behaves as $P$, but produces extra information about causality and QoS

Approach:
- Events in ORC are site calls (and returns) and publications (including intermediate ones)
- The idea is to instrument each event $e$ with causal and QoS additional information: $(e, \text{pre}(e), q(e))$
Causality tracking as a basis for QoS computation

Original program P
("The winner is " + x) <x< (Prompt("?")) | Prompt("?"))

Transformed program P'
(x>(vx,_)>
("The winner is " + vx,("The winner is ",[]):[x]))
<x< ((("Prompt",[])>u1>Prompt("?">w1>(w1,[u1]))) | (("Prompt",[])>u2>Prompt("?">w2>(w2,[u2])))
Example of response times

("The winner is Claude", [("The winner is ", []),
("Claude", [("Prompt", [])]))}

"The winner is ", d0

"Prompt ", d1

"Claude", d2

"The winner is Claude", max(d0,d1+d2)
The ORC calculus

\[ \begin{align*}
\nu & \in \text{Value} \\
\mathbf{x}, \mathbf{x}_1, \ldots, \mathbf{x}_n & \in \text{Variable} \\
f, g & \in \text{Expression} \\
\end{align*} \]

\[ \begin{align*}
\nu & \mid x \mid x(x_1, \ldots, x_n) \mid f \mid g \mid \\
f \prec \mathbf{x} \succ g \mid f \prec x \prec g \mid f ; g \mid \\
\text{def } x(x_1, \ldots, x_n) = fg
\end{align*} \]
Transformation rules for causality

\[ [v]_c \rightarrow (v, c) \]
\[ [x]_c \rightarrow (v, \{x\} \cup c) < (v, _) < x \{ - v \text{ fresh} - \} \]
- function call
\[ [x(x_1, \ldots, x_n)]_c \rightarrow x((v_1, c_1 \cup c) < (v_1, c_1) < x_1, \ldots, (v_n, c_n \cup c) < (v_n, c_n) < x_n) \]
- site call
\[ [x(x_1, \ldots, x_n)]_c \rightarrow ((x, \bigcup_{1 \leq i \leq n} c_i \cup c) > u > x(v_1, \ldots, v_n) > (v', Y) > (v', Y \cup \{u\})) < (v_1, c_1) < x_1 \ldots < (v_n, c_n) < x_n \]
\[ [f | g]_c \rightarrow [f]_c | [g]_c \]
\[ [f > x > g]_c \rightarrow [f]_c > x > [g]_{\{x\}} \]
\[ [f < x < g]_c \rightarrow [f]_c < x < [g]_c \]
\[ [\text{def } x(x_1, \ldots, x_n) = fg]_c \rightarrow \text{def } x(x_1, \ldots, x_n) = [f]_c [g]_c \]
The otherwise operator: tracking halts

All events inside the scope of the $f;g$ operator are recorded in a buffer. When $f$ halts, they form the causes of the halting event $h$, cause of $g$.

```
val trace = Buffer()
def max([], u) = trace.put(u)
def max(m : ms, (x, px)) = if member(m, px) then signal
    else trace.put(m) ≫ max(ms, (x, px))
def record(u) = trace.getAll() >ms> max(ms, u)
def track(u) = (u, record(u)) > (y, _) > y
```

```
[f ; g]c  →  [[f]c ; track("h", trace.getAll())) >x> [[g]{x}
   −x  fresh
```
Extension with QoS

Consider the general case of composite QoS domain, which is partially ordered

$$Q = (\mathbb{D}_q, \leq_q, \oplus_q)$$

- Each event is equipped with a QoS increment value
  $$e = ((v, q, Q), pre(e))$$

- The associated QoS may be recursively computed using the causal past
  $$Q(e) = \left( \bigvee_{e' \in pre(e)} Q(e') \right) \oplus q(e)$$
  synchronizing the causes
  increment
Extending ORC with a best QoS pruning operator: solving conflicts by QoS competition

**New pruning operator** Demands in general to wait for all the first publications of $g$

\[
f < \times <_q g
\]

\[
Q = (D_q, \leq_q, \oplus_q, \triangleleft_q)
\]

- Direct conflicts are recorded with the event

\[
e = ((v, q, Q) \text{pre}(e), \text{directconflicts}(e))
\]

- Used in the QoS computation

\[
Q(e) = \left( \left( \bigvee_{e' \in \text{pre}(e)} Q(e') \right) \oplus_q q(e) \right) \triangleleft (Q(e') \mid e' \in \#(e))
\]
Implementation: the principles

- Separate description of the composite QoS domain and its related algebra
- The original ORC program is then weaved (instrumented) with the QoS description
- Publications of the weaved program contain the QoS information
- Use of XML/OIL intermediate form
  - This form is parsed and printed using SCALA functions
  - Rules are implemented using ORC expressions and sites implemented in SCALA
  - The ORC engine executes the transformed OIL program
SLA description in ORC

```scala
def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
def class InterQueryTime()=
  def QoS(siteX) =
    val s = {. r = Ref(0), c = Channel() .}
    val curTime = Rclock().time()
    s.r? |> (s.c.put(curTime-p) | s.r:=(curTime)) >>
    Dictionary() >siteX> siteX.InterQueryTime := s
  def QoSCompare(it1,it2) = it1 >= it2
  def QoSCompete(it1,it2) = bestQoS(QoSCompare,[it1,it2])
  stop

def class ResponseTime() =
  def QoS(siteX,d) = Rclock().time()-d + 100 >q> q
  def QoSPlus(rt1,rt2) = rt1+rt2
  def QoSCompare(rt1,rt2) = rt1 <= rt2
  def QoSCompete(rt1,rt2) = bestQoS(QoSCompare,[rt1,rt2])
  def QoSVee(rt1,rt2) = max(rt1,rt2)
  stop

def class Cost() =
  def QoS(siteX,c)=
    val s = Ref([])
    s? |> (xs,ys) >q> (xs,ys) >> Dictionary() >siteX> siteX.Cost := s
  def QoSPlus(c1,c2) =
    def Oplus([],[]) = []
    def Oplus(xs,y:ys) = (x+y):Oplus(xs,ys)
    Oplus(c1,c2)
  def QoSCompare(c1,c2) =
    def Compare([],[]) = true
    def Compare(xs,ys) = (x <= y) && Compare(xs,ys)
    Compare(c1,c2)
  def QoSCompete(c1,c2) = bestQoS(QoSCompare,[c1,c2])
  def QoSVee(c1,c2) =
    def Vee([],[]) = []
    def Vee(xs,ys) = max(xs,ys)
    Vee(c1,c2)
  stop
```
Soft contracts

QoS contracts cannot rely on hard bounds

▶ Why not a soft bound, covering 95% of the cases?
▶ Unfortunately, such contracts do not compose
▶ Idea: a contract is a probability distribution
Probabilistic contracts

- The contract consists of a probability distribution
- Probas compose well:
  - use Max-Plus probabilistic algebra if the control is deterministic
  - otherwise run Monte-Carlo simulations
- QoS distributions can result
  - from contracts
  - from measurements
To make it practical, we can define probabilistic contracts by specifying only a finite set of quantiles (expressible in WSLA)
The specified contract $F(x) = Pr(\delta \leq x)$ (probability density)

A distribution $G(x)$ breaching the contract, meaning that

$\neg (G \succeq_S F)$, where $\succeq_S$ denotes stochastic dominance ($\forall x, G(x) \geq F(x)$)

$G$ is unknown: it is observed. How to perform on-line detection of the contract violation?
On-line detection

- Actual test running with $t$: $\sup_x [F(x) - G_{[t, t+N]}(x)] \geq \lambda$
- $G_{[t, t+N]}(x)$ empirical distribution function based on $[t, t+N]$

Calibration is performed by bootstrapping:

1. Build large training data set (Monte-Carlo simulation of contract distribution)
2. Resample it many times by selecting $N$–size trials
3. Tune $\lambda$ so that 95% of trials are accepted
Conclusion

Web services orchestrations or choreographies are a world of contracts

- SLA: function & QoS jointly
- The paradigm of contracts (composition, monitoring, reconfiguration)
- Novel issues
  - Function: workflow & data
  - QoS: monotonicity
  - QoS: soft contracts
Conclusion

We have proposed a comprehensive approach

▶ QoS algebra
▶ Probabilistic soft contracts
▶ Contract composition
▶ Statistical contract monitoring
▶ Reconfiguration?

A mix of techniques

▶ Formal concurrent models for orchestrations (ORC, Petri nets)
▶ Monte-Carlo simulation
▶ Bootstrap methods from statistics
Conclusion

Thank you
References I

Albert Benveniste, Claude Jard, Ajay Kattepur, Sidney Rosario, and John A. Thywissen.
Qos-aware management of monotonic service orchestrations.
Under submission, 2012.

Sidney Rosario, Albert Benveniste, and Claude Jard.
Flexible probabilistic qos management of orchestrations.
*International Journal of Web Services Research, 2, 2010.*

Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard.
Probabilistic qos and soft contracts for transaction-based web services orchestrations.
References II

Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard.
Pairwise testing of dynamic composite services.
In *6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS)*, SEAMS ’11, pages 138–147, New York, NY, USA, 2011. ACM.

Ajay Kattepur.
Importance sampling of probabilistic contracts in web services.

Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard.
Variability modeling and qos analysis of web services orchestrations.
References III
