Towards Data-Aware QoS-Driven Adaptation for Service Orchestrations

March 2010

Dragan Ivanović
Manuel Carro
Manuel Hermenegildo
Pedro López-Garcia
Edison Mera

TR Number CLIP 5/2009.1

Technical Report Number: CLIP 5/2009.1
March,2010

Authors

Dragan Ivanović
idragan@clip.dia.fi.upm.es
Computer Science School
Universidad Politécnica de Madrid (UPM)

Manuel Carro
mcarro@fi.upm.es
Computer Science School
Universidad Politécnica de Madrid (UPM)

Manuel Hermenegildo
herme@fi.upm.es
Computer Science School
Universidad Politécnica de Madrid (UPM)
and IMDEA Software, Spain

Pedro López
pedro.lopez@imdea.org
Computer Science School
Universidad Politécnica de Madrid (UPM)
and IMDEA Software, Spain

Edison Mera
edison@fdi.ucm.es
Facultad de Informática
Universidad Complutense de Madrid (UCM)

Acknowledgements

This research has been funded by the EU 7th. FP NoE S-Cube 215483, and partially by FET IST-231620 *HATS*, EUREKA 06042 *ES_PASS*, MICINN TIN-2008-05624 *DOVES*, MIN FIT-340005-2007-14, and CM project P2009/TIC/1465 (*PROMETIDOS*).

Abstract

Several activities in service oriented computing can benefit from knowing properties of a given service composition ahead of time. We will focus here on properties related to *computational cost* and *resource usage*, in a wide sense, as they can be linked to QoS characteristics. In order to attain more accuracy, we formulate computational cost / resource usage as *functions on input data* (or appropriate abstractions thereof) and show how these functions can be used to make more informed decisions when performing composition, proactive adaptation, and predictive monitoring. We present an approach to, on one hand, automatically synthesize these functions from orchestrations and, on the other hand, to effectively use them to increase the quality of non-trivial service-based systems with data-dependent behavior. We validate our approach by means of simulations with runtime selection of services and adaptation due to service failure.

Keywords: Service Orchestrations, Resource Usage Analysis, Data Awareness, Monitoring, Adaptation.

Contents

1 Introduction

Service Oriented Computing (SOC) is a well-established paradigm which aims at expressing and exploiting the computation possibilities of loosely coupled systems which interact remotely. Such systems expose themselves via service interfaces whose description may include operation signatures, descriptions of behavior, and others, while the implementation is completely hidden. Services can be combined to accomplish more complex tasks through *service compositions*, which are usually expressed using either a general-purpose programming language or languages designed to express business processes and compositions [4, 8]. These compositions can in turn expose themselves as full-fledged services.

One distinguishing feature of SOC systems is that they are expected to be active during long periods of time and span across geographical and administrative boundaries. These characteristics require having monitoring and adaptation capabilities at the heart of SOC. Monitoring compares the actual and expected system behavior. If a too large deviation is detected, an adaptation process (which may involve, e.g., rebinding to another provider of a service) may be triggered. When deviations can be predicted before they actually happen, both monitoring and adaptation can act ahead of time (being termed, respectively, *predictive* and *proactive*), performing prevention instead of healing.

Detecting deviations requires a behavioral model, which is used to check the current behavior or to predict a future behavior. Naturally, the more precise a model is, the better adaptation / monitoring results will be achieved. In this paper we will develop and evaluate models which, based on a combination of static analysis and actual run-time data, increase accuracy by providing upper and lower approximations of computational cost / resource usage measures which can be related to QoS characteristics. For example, the number of service invocations can be related to execution time when information about network speed is available.

2 Computation Cost Analysis and Services

Computational cost analysis aims at statically determining the computational cost (in terms of, e.g., execution steps or number of instructions) of a given algorithm for some input data. Tools to perform this kind of analysis have been developed in the field of programming languages.

However, to the best of the authors' knowledge, no similar work exists for SOC, although several approaches to automatically deriving QoS characteristics for compositions have been proposed [3, 2]. While these have much in common with our proposal, they do not treat data operations or relate QoS estimation with the characteristics of input data. Instead, some execution characteristics (e.g., number of iterations in a loop) are often either fixed or modeled statistically. Also, aggregating QoS characteristics of service compositions exposed as services is often not done. Some proposals [1] aim at performing global optimization, but still ignore data-related issues. Our proposal addresses both dimensions (global information and data-sensitivity) while still aiming at a completely automatic analysis.

2.1 A Motivating Example

We illustrate the relevance of taking actual data into account when generating QoS expressions for service compositions with a motivating example.

Fig. 1 shows a fragment of a (stylized) car part reservation system. A part Provider serves its

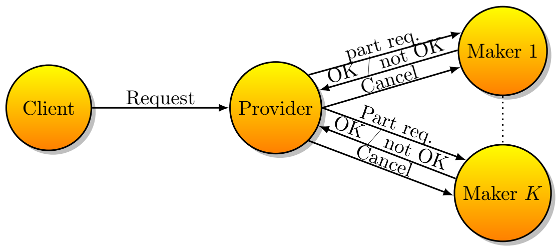

Figure 1: Simplified car part reservation system.

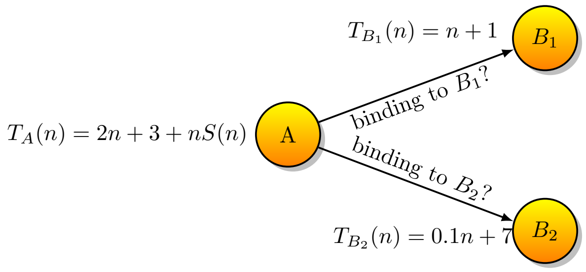

Figure 2: Invoking other services.

Client by reserving a number of part types from a pool of part Makers. The protocol only allows the Provider to reserve one part type per service invocation to a Maker. An invoked Maker replies `ok` if the part type is available and `not ok` otherwise; in this case the Provider goes to another Maker. If no Maker can reserve some car part type, the Provider cancels all previously reserved part types with a `cancel` message. Since every service invocation takes some time to complete, the number of car part types impacts the total time that Provider needs to complete a reservation for Client. Thus, a precise model of the time needed by Provider should take into account the *Request*, and more accurate time estimations should be expressed as functions on properties (e.g., number of types) of the incoming *Request* message.

2.2 Computational Cost of Service Networks

The function which results from the analysis of computational cost depends on the internal logic of the service composition (the Provider, in our example), but also on the behavior of the invoked services (the Makers), as they may, in turn, send additional messages which add to the global count.

Fig. 2 depicts this scenario in some detail. The input message is abstracted in this example as a parameter n (i.e., the number of car part types in our example) on which some measure of computational cost depends. The cost of service A is $T_A(n)$. As A invokes n times another service, (represented by a generic S), for which B_1 and B_2 are two candidates with different computational cost, its overall computational cost depends as well on which service is selected to perform the composition. Using the $T(n)$ values from Fig. 2, the computational cost corresponding to these two options would be:

$$\begin{array}{lll} T_{A_1}(n) = 2n + 3 + n(n+1) & = n^2 + 3n + 3 & \{\mathrm{AB}_1\} \\ T_{A_2}(n) = 2n + 3 + n(.1n + 7) & = 0.1n^2 + 9n + 3 & \{\mathrm{AB}_2\} \end{array}$$

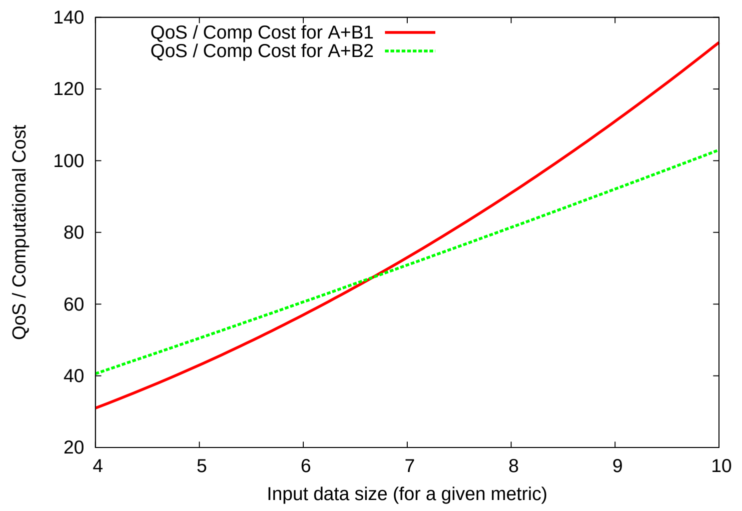

Figure 3: Computational cost, services AB_1 and AB_2.

and to decide between B_1 or B_2, T_{A_1} and T_{A_2} have to be compared (Fig. 3). This opens up the possibility of taking into account the size n of the data to select a configuration depending on the expected usage, and it requires information about B_1 and B_2 in order to automatically work out the resulting overall computational cost.

The computational cost-related information for B_1 and B_2 can be made available in much the same way as other service-related information (e.g., interfaces or XML schemes) is published. It needs to include, at least, the expected computational cost (preferably as a function of input data characteristics) and (possibly) the relationship between the sizes of the input and output data for every operation in the interface. The availability of these descriptions can make it possible to *automatically* work out T_{A_1} and T_{A_2} to compare them. In turn, A should publish the information it synthesizes, so that it can then be used by other compositions. In our view, this repeated process of synthesis, comparison, and publishing, is a step towards simultaneously achieving true dynamicity and optimal selection in the creation and adaptation of service networks.

Note that these abstract descriptions do not compromise the privacy of the implementation of the service being described, as they act as a high-level contract on the behavior of the service. Besides, in an open ecosystem of services, those which publish such descriptions would have a competitive advantage, as they make it possible for customers to make better decisions on which services to bind to.

Given a service A, if we assume that any services it invokes have a constant computational cost $T_{B_i}(n) = 1$, then the computational cost obtained for A measures how much its structure alone contributes to the total computational cost. We have termed this the *structural computational cost* of a service, and it will be used later as an approximation of the real computational cost.

Two key questions are: to which point functions expressing the cost of the computations are applicable to determining QoS, and to which point these functions can be automatically (and effectively) inferred for service compositions.

2.3 Approximating Actual Behavior

The computational cost measures we will deal with count relevant *events* which are deterministically related to the input data: processing steps, number of service invocations, size of the messages, etc. To infer such computational costs we follow the approach to resource analysis of [7] which, given data on how much a few selected basic operations contribute to the usage of some resource, tracks how many times such basic operations are performed through loops and computes the overall consumption of the resource for a complete computation. Since the number of loop iterations typically depends on the input, the overall consumption is given as a function that, for each input data size, returns (possibly upper and lower bounds to) the overall usage made of such resource for a complete computation.

Different higher-level QoS characteristics can then be derived from these functions: execution time can be approximated by aggregating the number of basic activities executed and the number of invocations, and multiplying them by an estimation of the time every (type of) activity and invocation takes; availability of a composed service can be expressed as the product of the availability of the services it invokes (assuming independence between them) and, therefore, the availability of the composition will depend on which services are invoked and how many times they are invoked, which in turn depends on the input data.

Estimations of the time used, availability, etc. of basic components are approximate and they thus introduce some noise which also makes the derived QoS functions approximations. However, because they are functions on input data they are likely to predict more accurately the behavior for a given input than a global statistical measure (we return to this later). Besides, for cases where comparison between two different QoS functions (and not their absolute value) is relevant, as in Fig. 2, the noise introduced can be expected to mutually cancel to some extent.

2.4 Upper and Lower Bounds

Automatically inferred computational cost functions can sometimes be exact, but in general only safe upper and lower bounds can be generated. These are guaranteed to be smaller than or equal to (resp. greater or equal) the function they approximate. This can be traced back to limitations of the static analysis, to the actual function depending on more parameters than, e.g., data size, and others. When these bound functions are combined with estimations to determine QoS from computational cost functions, data-aware approximations of the actual bounds are created.

While this may seem to be a disadvantage when it comes to predicting future behavior, upper / lower bounds of the actual computational cost are actually useful to *ensure* that some QoS characteristic is met, because it falls above / below the predicted threshold. As an example, Fig. 4 portrays upper and lower bound computational cost functions for two compositions for some QoS characteristic which depends on input data. Depending on the QoS meaning, we may want to make sure that we stay above or below some value. The former case needs to consider the lower bound and, conversely, the latter requires considering the upper bound. Note also that, in the example portrayed in the figure, which service will give better results clearly depends on the actual data size at run-time.

Comparing data-aware approximating functions with the probabilistic approximations used in many approaches to QoS-driven service compositions can be illustrative. Average approximations which summarize QoS characteristics in a single point clearly cannot provide behavior guarantees, as they do not provide ranges for maximum and minimum values, and they do not

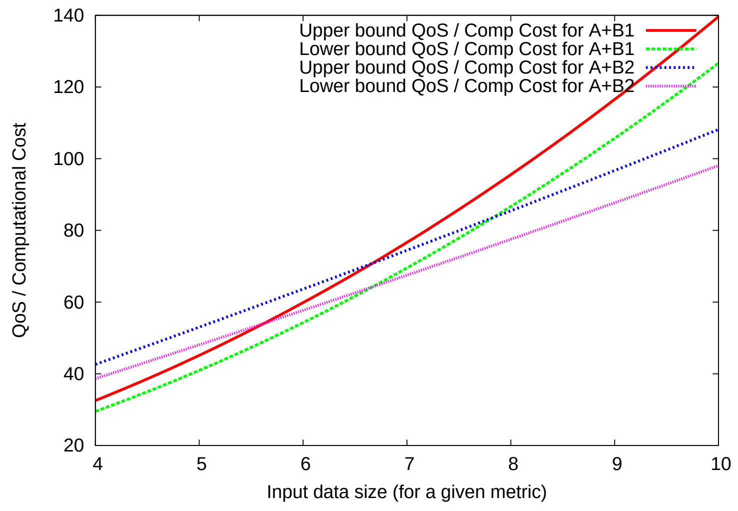

Figure 4: Using upper/lower bounds.

take data ranges into account. The statistical approach can be extended in two directions: an interval can be used to represent the maximum and minimum of the QoS, measured across all the possible input data range. But it is a coarse approximation, as it does not take into account any correlations of the QoS with input data. The other direction corresponds to using a function which, for every possible input data, represents some average value of the characteristic. This can be more precise than using a single point, but again it does not provide any bounds (not even approximate) for the QoS values.

Combining these two extensions boils down to using functions over input data which represent upper and lower bounds, and which are transformed into QoS functions by appropriately plugging in actual execution characteristics, as suggested in Section 2.3. While the results are not strictly safe, we claim that these QoS bounds can be used to predict whether the future history will stay within some predefined limits with better accuracy than just a static point, static bounds, or an average. In any of the latter cases, less information than with the upper / lower bound approximate functions is provided, so any decision will be less informed.

3 Analysis of Orchestrations

Our approach is based on translating process definitions into a language for which automatic computational cost analysis tools are available. We will now give details on this process, sketched in Fig. 5.

3.1 Overview of the Translation

Our input languages are a subset of BPEL 2.0 for the process definitions and WSDL for the associated meta-information. These are translated into an intermediate language (Table 1)

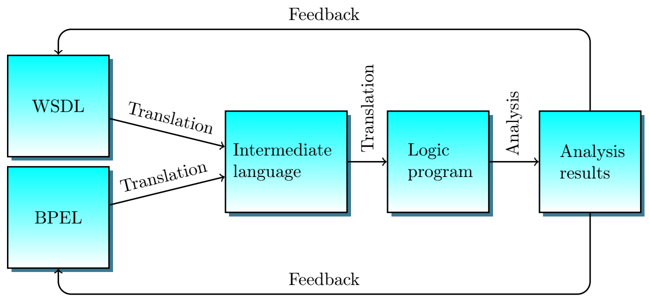

Figure 5: The overall process.

	Declarations and definitions
Complex type definition	`:-struct`(*QName*, *Members*).
Port type definition	`:-port`(*QName*,*Operations*).
External service	`:-service`(*PortName*,*Operation*, {*TrustedProperties*}).
Service definition	`service`(*Port*, *Operation*, *InMsg*, *OutMsg*)`:-`*Activity*.
	Activities
Variable assignment	*Var* `<-` *Expr*
Service invocation	`invoke`(*PortName*, *Operation*, *OutMsg*, *InMsg*).
Reply and exit	`reply`(*OutMsg*)
Sequence	$Activity_1$ `,` $Activity_2$
Conditional execution	`if`(*Cond*, *ActThen*, *ActElse*)
While loop	`while`(*Cond*, *Activity*)
Repeat-until loop	`repeatUntil`(*Activity*, *Cond*)
For-each loop	`forEach`(*Var*, *Start*, *End*, *Activity*)
Scope	`scope`(*VarDecl*, *ActivityList*)
Scope fault handler	`handler`(*FaultName*, *Activity*)
Parallel flow	`flow`(*LinkDecl*, *Activities*)
Activity in a flow	`float`(*Attributes*, *Activity*)

Table 1: Abstract orchestration elements.

which can also be used to cover other orchestration languages.[1] This intermediate representation is then translated into the *Ciao* logic programming language [6], which includes assertions to express types and input / output modes for arguments, as well as resource definitions and functions describing resource usage bounds. The resulting logic program is then analyzed by the CiaoPP tool [5], which is able to infer upper and lower bounds for computational costs [7], among other analyses.

A BPEL process definition is translated into a service definition which associates a port name and an operation with an activity that represents the orchestration body. BPEL processes forming a service network are translated into predicates which call each other to mimic service invocations.

The intermediate language can describe namespace prefixes, XML schema-derived data types

[1]Although it currently models mainly BPEL constructs.

```
:- regtype 'factory->resData'/1.
'factory->resData'('factory->resData'(A, B, C)):-
     num(A), num(B), list(C, 'factory->partInfo').

:- regtype 'factory->partInfo'/1.
'factory->partInfo'('factory->partInfo'(A, B)):-
    atm(A), atm(B).
```

Figure 6: Translation of types.

for messages, service port types, and also known properties of external services of interest to the analysis (when such services are not analyzed). The activities supported by the intermediate language include generic constructs (assignment, sequences, loops...) and specific constructs to model orchestration workflows: `flow`, `float`, `scope`/`handler`, and `invoke`. `flow` corresponds to the similarly named BPEL activity, while the `float` construct annotates an activity within a `flow` with a description of outgoing links and their values, join conditions based on incoming links, and a specification of the behavior in case of a join failure.

A relevant observation regarding the translation is that it does not need to follow strictly the operational semantics of the orchestration language: it has to capture enough of it to ensure that the analyzers will infer correct information while minimizing precision loss due to the translation. Despite this, in our case the translated program is executable, and mirrors quite closely (but not exactly) the operational semantics of the BPEL process under analysis.

3.2 Restrictions on Input Orchestrations

Our analysis is restricted to orchestrations which follow a *receive–reply* pattern, where all activities start after receiving an initial message and finish by dispatching either a reply or a fault notification. Additionally, we currently do not support the analysis of stateful service callbacks using correlation sets or WS-Addressing schemes. In the future we plan to relax both restrictions by identifying orchestration fragments that correspond to the *receive–reply* pattern.

In our intermediate language, we support a variant of the `scope` construct, which introduces local variables and fault / compensation handlers. We do not fully support compensation handlers, which in BPEL "undo" the effects of a successfully completed scope using snapshots of variables recorded at successful completion of the scope. Except for recording snapshots, compensation handlers can be treated as pseudo-subroutines on a scope level, and inlined at their invocation place.

3.3 Type Translation and Data Handling

The simple types in XML schemata are abstracted as three disjoint types: `number`s, strings (translated into `atom`s), and `boolean`s. Complex XML types are translated into predicates specifying how the type is built. Fig. 6 shows the translation corresponding to a fragment of the reservation scenario in Section 2.1. The type named `'factory->resData'` is a structure with three fields: two numbers and a list of elements of type `'factory->partInfo'`. Each of these elements is in turn a structure with two fields (atoms).

The accepted expression language is a subset of XPath which allows node navigation only along the descendant and attribute axes. This ensures that navigation is statically decidable and XML structures can be deforested to pass the addressed components as sepa-

A	Translation of $T([A\|R], \eta, V)$	
`empty`	$T(R, \eta, V)$	(*Empty action*)
$A_j,\ A_k$	$T([A_j, A_k\|R], \eta, V)$	(*Sequence*)
$\texttt{reply}(v)$	$V = \texttt{reply}(\eta(v))$	(*End of orchestration*)
$\texttt{throw}(f)$	$V = \texttt{fault}(f)$	(*No fault handler*)
	$T([H], \eta, V)$	(*Insert fault handler*)

Table 2: Inline translations.

A	Translation of $T([A\|R], \eta, Y)$
v `<-` e	$a(\eta, Y) \leftarrow E(e, \eta, X),\ T(R, \eta[X/v], Y)$
$\texttt{invoke}(p, o, v, w)$	$a(\eta, Y) \leftarrow s_{p:o}(\eta(v), Z),$ $(Z = \texttt{fault}(F) \rightarrow T([\texttt{throw}(F)], \eta, Y)$ $;\ Z = \texttt{result}(X) \rightarrow T(R, \eta[X/w], Y))$
$\texttt{if}(c, A', A'')$	$a(\eta, Y) \leftarrow C(c, \eta),\ !,\ T([A'\|R], \eta, Y)$ $a(\eta, Y) \leftarrow T([A''\|R], \eta, Y)$
$\texttt{while}(c, A')$	$a(\eta, Y) \leftarrow C(c, \eta),\ !,\ T([A', A], \eta, Y)$ $a(\eta, Y) \leftarrow T(R, \eta, Y)$
$\texttt{scope}(D, A'_H)$	$a(\eta, Y) \leftarrow T([A'_H], \eta[D], Z),$ $(\texttt{var}(Z) \rightarrow T(R, \eta, Y)$ $;\ Z = \texttt{fault}(F) \rightarrow T([\texttt{throw}(F)], \eta, Y)$ $;\ Y = Z)$

Table 3: Translation into predicates.

rate arguments when necessary to improve analyzer accuracy. For example, the expression `'$req.body/item[1]/@qty'` in the intermediate language refers to the attribute `qty` of the first `item` element in the `body` part of a message stored in variable `req`. A set of standard XPath operators and basic functions, such as `position()` and `last()`, are supported.

3.4 Basic Service and Activity Translation

An orchestration that implements operation o on port p is translated into a Horn clause

$$s_{p:o}(X, Y) \leftarrow T([A], \eta, Y).$$

where X and Y correspond to the initial message and the final reply and T corresponds to the translation of a list of activities (in this case just A, the body of the orchestration). η is an environment that maps orchestration variables to logical variables, which initially just maps the input message to X. New orchestration variables may be introduced with the `scope` construct. On exit, Y can be bound to either $\texttt{reply}(R)$, where R is the contents of the reply message, or $\texttt{fault}(F)$, where F is a fault identifier.

The translation operator T accepts a list of activities and produces a Prolog goal.[2] Then $T([\,], \eta, V) = \texttt{true}$ (nothing left to translate); otherwise the case is $T([A|R], \eta, V)$ and is driven by the structure of A (Table 2). The `empty` activity is skipped. A sequence of activities is unfolded and translated one by one. A $\texttt{reply(}v\texttt{)}$ unifies the result V with the value of the reply v in the current environment. If `throw` is executed in the scope of a fault handler `H`, it is executed; otherwise the result is unified with the fault identifier.

In more complex cases (Table 3), each activity is translated as a call to a predicate. A variable assignment v `<-`e generates a goal that evaluates e in η and unifies its result with variable X; the remaining activities R are translated with η updated with the new binding $[X/v]$. `Invoke`

[2]Following Prolog notation an empty list is written [] and a list with head A and tail R is written $[A|R]$.

```
<sequence>
  <while name='a_13'>
    <condition>$i>0</condition>
    <scope>
      <assign name='a_14'>
        <copy><from>$i - 1</from><to variable='i'/></copy>
      </assign>
      <assign name='a_15'>
        <copy><from>$resp.body/factory:part[$i]</from>
          <to variable='p'/></copy>
      </assign>
      <invoke name='a_16' portType='factory:sales'
        operation='cancelReservation' inputVariable='p'
        outputVariable='r'/>
    </scope>
  </while>
  <throw faultName='factory:unableToCompleteRequest'/>
</sequence>
```

(a) A BPEL code fragment

```
while( '$i>0', (                                  % a_13
  '$i' <- '$i␣-␣1',                              % a_14
  '$p' <- '$resp.body/factory:part[$i]',          % a_15
  invoke( factory:sales,cancelReservation,'$p','$r') % a_16
)),
throw( factory:unableToCompleteRequest)
```

(b) The intermediate representation.

```
a_13(A,B,C,D,E):- % ($i,$p,$resp.body/factory:part,$r,Y)
       A>0, !, a_14(A,B,C,D,E).
a_13(A,B,C,D,E):-
       E=fault('factory->unableToCompleteRequest').

a_14(A,B,C,D,E):-
       F is A-1, a_15(F,B,C,D,E).

a_15(A,B,C,D,E):-
       nth(A,C,F), a_16(A,F,C,D,E).

a_16(A,B,C,D,E):-
       'service_factory->sales->cancelReservation'(B,F),
       (  F=fault(G) -> E=fault(G)
       ;  F=reply(H) -> a_13(A,B,C,H,E)).
```

(c) Translation into logic program.

Figure 7: Translation example.

is similar, but it calls the target service predicate to obtain the result. `if` and `while` encode their condition with a call to a predicate C and a cut.

A `scope` is translated by nesting the translation of the activity/fault handler A'_H within updated environment $\eta[D]$, followed by a check for completion or faults. Faults within the scope are handled by H, and outgoing faults are rethrown. `flow` is translated similarly to `scope`, but without actually parallelizing the execution, since we are interested in the computational cost of the `flow` regardless of the number of threads. Links are modeled as Boolean variables, and dependent activities are sequenced to respect conditions on incoming/outgoing links. Dead-path elimination is supported.

3.5 A Translation Example

A translation example is presented in Fig. 7. Subfigure (a) is a BPEL fragment of an orchestration, (b) is the corresponding intermediate form, and (c) is the translation into a logic program. The orchestration traverses the list of part types to reserve from the external part

Resource	**With fault handling**		**Without fault handling**	
($n \geq 0$: input arg. value)	lower bound	upper bound	lower bound	upper bound
Basic activities	2	$7 \times n$	$5 \times n + 2$	$5 \times n + 2$
Single reservations	0	n	n	n
Cancellations	0	$n - 1$	0	0

Table 4: Resource analysis results for the group reservation service.

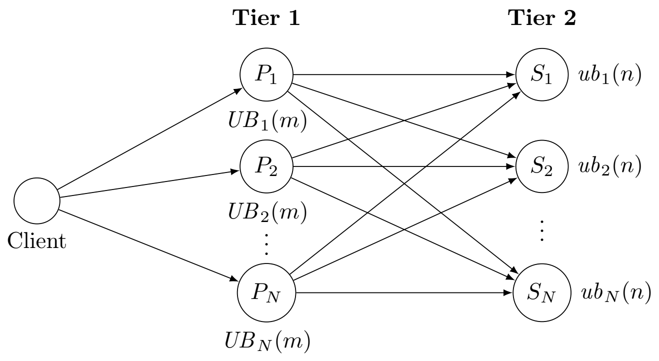

Figure 8: Two-tier simulation setting.

maker sales service.[3] If a fault arises, a fault handler tries to cancel already made reservations before signaling failure to the client. The figure shows just the `while` loop, which finishes with a `reply`.

The resource analysis finds out how many times external service invocations will be performed during process execution, from which deducing the number of messages exchanged is easy. The results for the complete orchestration are displayed in Table 4, where the estimated upper and lower bounds are expressed as a function of the input message.[4] We differentiate two cases: one in which fault-free execution is assumed, and another where fault handlers can be executed, which gives more cautious estimates. These two cases were obtained by turning on or off the generation of Prolog code for fault handling –the last part of Fig. 7 (c).

4 An Experiment in Adaptation

To validate our approach, we performed a simulation to study the effectiveness of applying data-aware computational cost functions to matchmaking and dynamic adaptation. We simulate a service network (Figure 8) where a client C selects among a set of providers P_i to reserve $n = 1..50$ sets of car parts. Each set consists of $M = 5$ different part types. The external client chooses one P_i which in turn chooses from among a set of part suppliers S_i, shared between all the providers. All P_i and S_i are known to be semantically equivalent, but vary in response time as the QoS attribute of interest. A P_i or S_i may fail with some probability p_f. When this happens, adaptation is triggered by searching for another (next-best) service from the pool.

The selection policies we have simulated are: random selection from the pool of candidates, fixed preferences, and data-dependent QoS prediction based on computational cost.

[3]Unlike in the example in Section 2.1, this code does not query different factories.
[4]The analyzer took 1.811 seconds to infer this information on a Intel Core Duo 2GHz machine with 2GB RAM and Darwin Kernel v10.2.0.

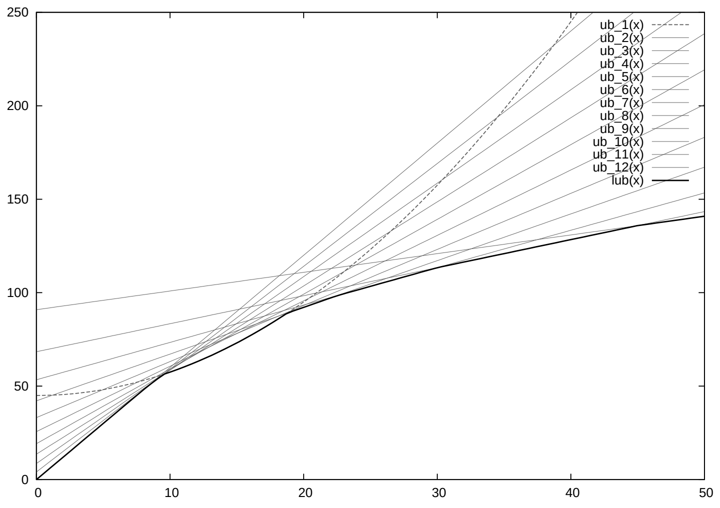

Figure 9: Upper bounds for computational costs.

In the last case, we select the best candidate taking into account its upper bound complexity (worst case behavior). Every service $S_i, 1 \le i \le 12 = N$, has a different upper bound cost function $ub_i(n)$ (portrayed in Figure 9), where n is the number of sets of a given part type. The bold line highlights the lowest upper bound among all the services for each n. $ub_i(n)$ measures the maximum number of messages exchanged by S_i as a function of the size of the incoming data. The computational cost for provider P_j is computed with the expression

$$UB_j(n) = E_{P_j}(n) + M + M \times ub^*(n)$$

which takes into account both the structural computational cost E_{P_j} (using the same family of curves in Figure 9) and that incurred by the services in the second layer: M times the cost ub_* of a service S_* selected for given n under the given selection policy.

Message exchanges are assigned a fixed time to convert them into execution time.[5] In a real scenario, this fixed amount of time can be updated as execution proceeds to reflect e.g. network state or system load.

The fixed preferences policy ranks services using the expected response time for some representative input; we chose $n = 12$. Therefore all queries whose data size is 12 are handled equally by both the fixed preferences and the data-dependent complexity cost approaches.

For each selection policy and for each n in the range 1..50, one hundred simulations are run and averaged. Each run performs matchmaking and simulates the execution of the selected service. Besides failures, the simulated number of outgoing messages in the run is (uniformly) randomly chosen between 60% and 100% of the upper bound, to model that sometimes this upper bound may not be needed. The time associated with every message exchange is padded with additional noise having a normal distribution to simulate the variations in the behavior of

[5]We are not taking into account the time associated to executing internal activities. The same technique used to infer the number of messages can be used to infer the number of activities of every type associated to some invocation, and can be accounted for in similarly to messages.

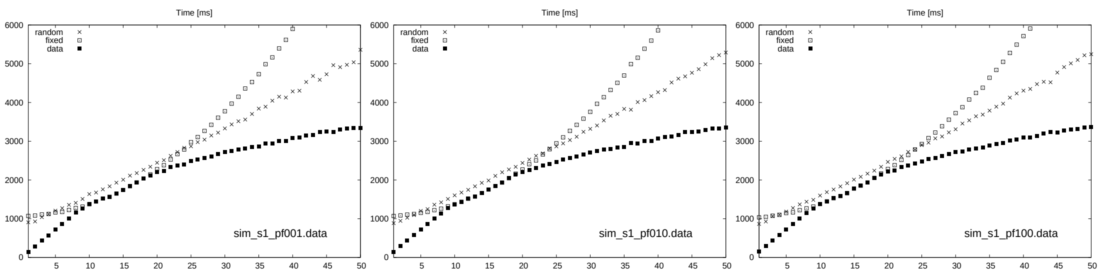

Figure 10: Simulation results for $\mathbf{p_f = 0.001, 0.01, 0.1}$ (left to right) and same noise distribution.

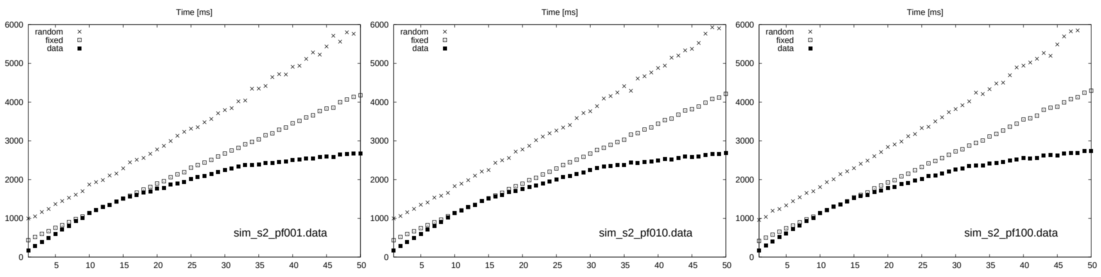

Figure 11: Simulation results for $\mathbf{p_f = 0.001, 0.01, 0.1}$ (left to right) and different noise distribution for each service.

the network.

Several sets of simulations with different time noise distribution parameters were performed, of which we have chosen two representative ones. In Fig. 10 all services have the same per-message average time (5 ms). In Fig. 11, services in both layers are assigned a different time per message whose average is in the range 4-8 ms. The figures show plots for the three selection policies, per each of the three failure probabilities used (left to right).

For most values of n, the data-dependent selection policy gives the best results and, notably, they feature a homogeneous and predictable behavior w.r.t. failure rates $p_f \in \{0.001, 0.01, 0.1\}$ and timing noise. Of course, it coincides with the selection made using the fixed preference policy for $n = 12$, where the fixed preferences were calculated. In an extended set of simulations (not appearing in this paper due to space constraints), the same behavior appears for even higher failure rates.

5 Conclusions

We proposed using data-aware computational cost functions to predict QoS adaptations and presented some preliminary results. We developed a translation-based scheme which, from an orchestration (in BPEL+WSDL), generates a (logic) program that can be analyzed by existing tools to automatically derive functions which are the upper and lower bounds of its computational cost. These functions are used to build more precise QoS estimations taking data characteristics into account which, in turn, can be used to, e.g., perform more precise predictive monitoring and proactive adaptation. We have reported on the results of a series of simulations where such data-aware QoS estimations were used to improve the efficiency of dynamic, run-time adaptation. The results are promising in that the data-aware adaptation always performs as well as any of the other policies studied, and in general gives better better results, even for cases with a very large variability in service behavior.

References

1. Mohammad Alrifai and Thomass Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *International World Wide Web Conference*, pages 881–890. ACM, April 2009.

2. J. Cardoso. About the Data-Flow Complexity of Web Processes. In *Int'l. WS on Business Process Modeling, Development, and Support: Business Processes and Support Systems: Design for Flexibility*, pages 67–74, 2005.

3. J. Cardoso. Complexity analysis of BPEL web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

4. D. Jordan et al. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, et. al., 2007.

5. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

6. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

7. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *Int'l. Conf. on Logic Programming*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.

8. Wil van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, 2006.