Towards Data-Aware Cost-Driven Adaptation for Service Orchestrations

facultad de informática

universidad politécnica de madrid

Dragan Ivanovic
Manuel Carro
Manuel Hermenegildo
Pedro Lopez-Garcia
Edison Mera

Authors

Dragan Ivanovic
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)

Keywords

Service Orchestrations, Resource Analysis, Data-Awareness, Monitoring, Adaptation

Abstract

Several activities in service oriented computing, such as automatic composition, monitoring, and adaptation, can benefit from knowing properties of a given service composition before executing them. Among these properties we will focus on those related to *execution cost* and *resource usage*, in a wide sense, as they can be linked to QoS characteristics. In order to attain more accuracy, we formulate execution costs / resource usage as *functions on input data* (or appropriate abstractions thereof) and show how these functions can be used to make better, more informed decisions when performing composition, adaptation, and proactive monitoring. We present an approach to, on one hand, synthesizing these functions in an automatic fashion from the definition of the different orchestrations taking part in a system and, on the other hand, to effectively using them to reduce the overall costs of non-trivial service-based systems featuring sensitivity to data and possibility of failure. We validate our approach by means of simulations of scenarios needing runtime selection of services and adaptation due to service failure. A number of rebinding strategies, including the use of cost functions, are compared.

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. In any case, such systems expose themselves as a service interface whose description may include operation signatures, behavioral descriptions, security policies, and other, while the implementation is completely hidden. Several services can be combined by calling the operations in their interfaces to accomplish more complex tasks than any of them in isolation through the process of *service composition*. Such compositions are usually expressed using either a general-purpose programming language or, alternatively, a language with an ad-hoc design aimed at expressing SOC compositions [11]. These compositions can in turn present themselves as full-fledged services.

One key distinguishing feature of SOC systems is that they are expected to live and be active during long periods of time and span across geographical and administrative boundaries. This brings the need to include monitoring and adaptation capabilities at the heart of SOC. Monitoring checks the actual behavior of the system and compares it with the expected one. If deviations are too large, an adaptation process (which may involve, e.g., rebinding to different services with compatible semantics and better behavior) may be necessary.[1] When deviations are detected before they happen (i.e., they are predicted), both monitoring and adaptation can act ahead of time (and they are then classified as *proactive*). Of course, the technology involved in proactive adaptation is more complex but also more interesting and useful, as it performs *prevention* instead of *healing*.

In any of these cases, it is necessary to have a model of the behavior of the composition against which the actual behavior is checked. Usual models try to capture for example service reliability or execution time, and use statistical analysis or log mining to find out values for these metrics. If the actual execution departs too much from the expected values, then a warning is issued. Additionally, if rebinding is needed in the course of an adaptation, then these characteristics can be used to select from among semantically equivalent candidate services. Needless to say, the more precise this model is, the better the adaptation / monitoring process can we expected to be.

In this paper we will be dealing with a particular kind of models: those which try to increase accuracy by, on one hand, taking into account actual run-time data and, on the other hand, giving always a correct value for the model at hand or, at least, a safe approximation. An example of such a value is the number of messages sent / received, which can be related to, for example, execution time (useful to determine some QoS characteristics) by assuming that data related to network speed is available, or to monetary cost if bandwidth usage has a cost (as, for example, in the case of short cell phone messages).

In this paper we will discuss how the ability to predict data-dependent execution characteristics can be of help in some situations (Section 2.1) and how the particular characteristics of SOC in relation with traditional computing paradigms can be taken into account (Section 2.3). As part of the needs of this architectural proposal, we will sketch how the models we propose can be automatically derived from the actual composition code (Section 3) and we will report on the results of a series of simulations which use data-enhanced models to drive a particular case of adaptation (Section 4).

[1] See the entries of *adaptation* and *monitoring* at `http://www.s-cube-network.eu/knowledge-model`.

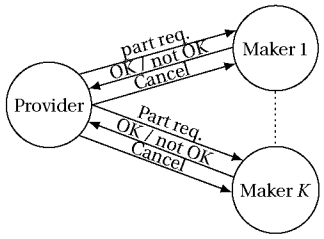


Figure 1: Simplified car part reservation system.

2 Cost Analysis and Service Networks

Cost analysis aims at statically determining the cost (in terms of execution time, execution steps, number of instructions, or other general resources) of a computation for some input data, given the code which expresses the computation. It has been studied for functional languages [15], logic languages [8, 7], object-oriented languages [2, 13] and it is also of use for worst-case execution-time analysis [16]. There are also approaches which aim at providing common libraries and representations to make cost analysis easier across several languages [12, 1].

To the best of the authors' knowledge, there has not been a similar study for SOC, although several approaches to automatically deriving QoS characteristics for compositions have been proposed [5, 4]. These have much in common with our proposal as they address the problem of working out aggregate costs for compositions. However, they do not fully treat data and do not relate cost estimation with actual input data sizes (they assume, for example, a statistically or otherwise fixed number of loop iterations). Also, aggregating QoS characteristics for complex networks using service compositions exposed as services (Section 2.3) is not treated. On the other hand, some proposals [3] aim at a global optimization, but ignore data-related issues. We will try to balance both dimensions (use of global information and data-sensitivity) while keeping the cost analysis automatic.

2.1 An Example

We illustrate with a simple and motivating example the benefit of taking actual data into account when generating QoS expressions for service compositions:

Example 1 *Figure 1 shows a simple car part reservation system. A car parts Provider needs to give a client a number of n (equivalent) car parts, and gets in touch with different part Makers' services to secure the shipment of these parts. The protocol is such that only a part can be reserved at a time from a maker using a service invocation. The Maker may answer* `OK` *if the part is available and* `not OK` *if it is not. In the latter case the Provider goes to the next Maker. If all the available Makers have been contacted and not all parts have been reserved, the Provider has to* `CANCEL` *all the reservations using the appropriate message. If some communication link is down or the maker service is not available, the communication is just not performed.*

We will assume that the Provider charges the client depending on the amount of CPU needed to fulfill a request (which we can approximate as the number of basic activities executed by the Provider) and that Makers charge the CPU provider per connection (which also should have an effect on the final price to the client). Additionally, both parameters should have an effect on the amount

of time that the Provider takes to answer to the client due to the number of messages necessary to process a request for car parts. Therefore, a more precise announcement of the cost or time for the Provider service should take into account the *size* of the requests made, i.e., the costs should be expressed as functions on the data used for the initial invocation. Additionally, there are two possible cases we may want to explore (which result in different behaviors): either the communications and the services are perfect (they do not fail) or there is the possibility that attempting to invoke the Maker fails.

The analysis is, often, non-trivial, even for these simple cases. The results depend, on one hand, on the internal logic of the service composition and, on the other hand, on the cost which each of the Makers charge the Provider for a given query. Section 3.7 shows how, for this particular example, we can automatically derive a number of cost-related functions which depend on data sizes (see Table 2). In that example, for the sake of simplicity, we have neglected the cost incurred by the Makers, but it should remain clear that in more complex examples these costs (which can in turn depend on input data – see Section 2.3) would generate more complex cost functions for the Provider — such as, e.g., quadratic.

We also want to highlight that, while in some cases these automatically generated cost functions are *exact* upper or lower bounds, in general, it can be expected that only *safe* upper and lower bounds of the actual costs are generated. These approximations arise either because of limitations of the static analysis, or because the actual cost depends on more parameters than data size, and, thus, an exact cost function based only on data sizes does not exist.

By *safe approximation* (safe upper and lower bounds) we mean that an upper bound (c.f., a lower bound) is always guaranteed to be bigger (c.f., smaller) than the actual cost function. While this may seem to be a disadvantage when it comes to predicting actual costs,[2] this upper or lower bounding of the actual cost is necessary when what is needed is to statically *ensure* that some QoS characteristic (e.g., from a contract) is met, or, conversely, to prove that some QoS characteristic will not be met.

It is illustrating to compare safe approximating functions with probabilistic approximations, used in many approaches to QoS-driven service compositions. Statistical approximations which summarize the cost characteristics in a single point, that is supposedly valid for all data within the input range, clearly cannot provide any behavior guarantee, as in general this point represents some kind of global average instead of a maximum or minimum. This can be extended in two directions: an interval can be used, where, in order for its bounds to be significant, they have to represent the maximum and minimum of the characteristic being measured across all the possible input data range. This is of course safe, but it is an overly gross approximation, as it does not take into account any correlation of the cost characteristic with the input data. The other direction corresponds to using functions which, for every input data, represent some average value of the characteristic. This can be more precise than using a single point, but it does not allow giving any guarantee. The combination of the two extensions proposed, i.e., the use functions which represent upper and lower bounds for different input data, makes it possible to provide more precise guarantees across the complete range of input data, and therefore allow, at least in principle, the possibility of making more informed service selections.

As an example, Figure 2 portrays the upper and lower bounds of two compositions for some QoS characteristics as a function of some input parameter. Depending on the meaning of these characteristics we may want to make sure that we minimize them (for example, if we want to exchange a small number of messages) or maximize them (if we want to increase the throughput of the system). The former case needs to consider the upper bound (as minimizing the upper bound the whole

[2]Note, however, that when the inferred upper and lower bounds coincide they are exact cost functions.

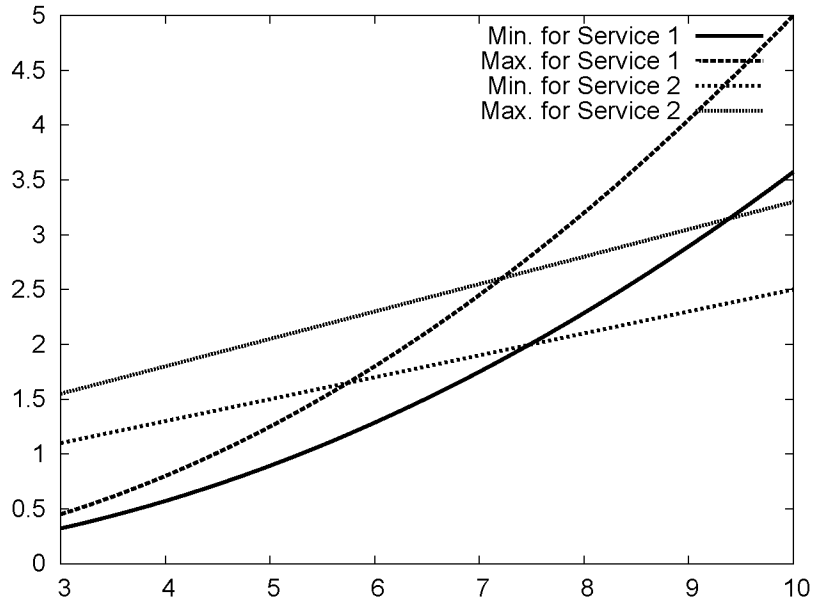


Figure 2: Upper and lower bounds for two services.

function is necessarily minimized) and, conversely, the latter requires considering the lower bound. According to Figure 2, selecting one or another service depends on the particular data size at hand.

2.2 Cost Functions Under Consideration

The type of cost characteristics we will take into consideration are based on counting a number of relevant *events*. To this end, we follow the approach to resource-oriented analysis of [14, 13, 12]. The fundamental idea is to specify how much some basic operations in a program contribute to the usage of some resource, and derive cost functions based on that specification for the whole program using global analysis techniques.

Higher level characteristics (expressed as compound cost functions) can be derived from these basic cost functions, which have a meaning on their own. For example, execution time can be built by aggregating the number of basic activities executed (for CPU time) and the number of messages exchanged taking into account the network latency and bandwidth. Functions built from upper bounds can be upper bounds as well (resp. lower bounds). Of course, if the aggregation of cost functions introduces noise (for example, by using inaccurate estimations of actual bandwidth), the resulting compound functions will not be accurate. However, as long as the noise is uniformly introduced in all involved functions, comparing aggregated functions should be sound.

Since inferring functions representing upper/lower bounds does not depend on what these functions exactly represent, and comparing them is also independent from their meaning, we will assume

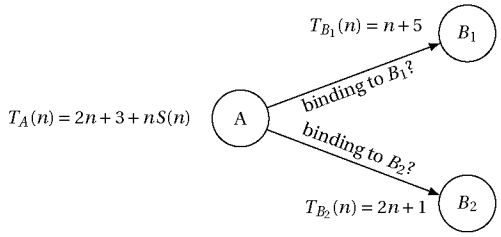


Figure 3: Invoking services with non-null cost.

in what follows that they represent generic costs which, in general, we want to minimize.

2.3 Costs for Service Networks

In the previous sections we dealt with the cost of a single composition under the assumption that the services it binds to do not contribute to the cost of the compositions. In general this is not so, and when the definition of these accessed services (B_i in Figure 3), which may be compositions themselves, is available, they can be analyzed together with the code of A to derive a global cost. If the code of some B_i is not available or, for some reason, the owner of that service does not want to reveal it, the cost function for A can still be inferred if at least B_i publishes its cost functions (and a description of how the sizes of its input and output messages are related, given as a data size function) so that the analyzer can use them directly instead of working them out. Note that publishing these cost and size functions should not compromise the confidentiality of the service B_i itself.

Assuming that cost functions are cumulative, an upper bound for the cost of A can be expressed, for the case of binding to only one service, in a form similar to

$$T_A(n) = E_A(n) + g(n)S(f(n))$$

where E_A is a *structural cost function* which accounts for the contribution of the code of A without taking into account the contribution of the services it may use, whose upper bound is summarized as $S(f(n))$. The function f represents the upper bound of the possible difference between the input data for A and that which is passed on to the invoked service, and g is an upper bound on the number of times S is invoked. The cost for a given composition comes from replacing S with the B_i corresponding to the selected service. This process may need to be repeated for the services used by A in order to generate a cost function which depends solely on the input parameters to A, but which is potentially different for every different binding of A to a service. For example, (the upper bound of) the costs corresponding to the composition A when binding to services B_1 and B_2 would be, respectively

$$\begin{aligned} T_A(n) &= 2n+3+n(n+5) = & n^2+7n+3 \quad & \text{for } B_1 \\ T_A(n) &= 2n+3+n(2n+1) = & 2n^2+4n+3 \quad & \text{for } B_2 \end{aligned}$$

Which one of them is bigger depends on the input data.

Note also that this process may have to be repeatedly applied down the stream of invoked services — i.e., B_i may be a composition invoking other services and may need performing a cost analysis to provide closed cost functions. This is a consequence of the dynamicity of service-based applications which is not usually found in traditional software: since the precise components of a given application can change dynamically, the cost functions of a composition can only be completely determined when this composition is completely known, including the exact services it binds to (or, at

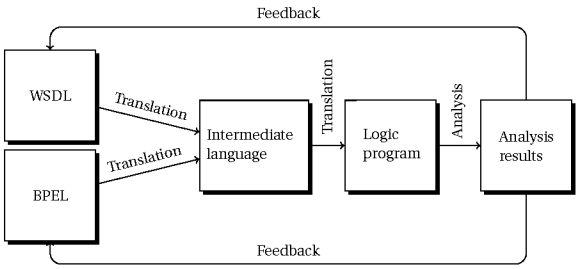


Figure 4: The overall process.

least, their associated costs and size relations between input and output data). Therefore, since the application can change dynamically, in order to be up to date the cost of the compositions affected by that change has to be recomputed — preferably in an incremental fashion in order not to waste resources.

A key question is how the functions expressing cost and data size relationships can be automatically and effectively inferred for service compositions. As discussed before, this has been studied previously, but the role of input data has not been satisfactorily (and safely) taken into account so far. We will devote the next section to presenting our approach. Note that we assume that there is a point where services do not invoke other services (i.e., they are leaves in an invocation tree) and their cost bounds are either determined using an approach similar to the one we will present in Section 3 or the ones in [14, 13]. Therefore, we will now focus on how cost functions can be inferred for a given service composition, with the understanding that they may be later subject to combination across a service network as previously shown.

3 Analysis of Orchestrations

Our approach is based on translating process definitions, via an intermediate language, to a logic program to be analyzed by existing tools (see Figure 4). In our case, the input language is a subset of BPEL 2.0 (for the process definitions – see Section 3.2) and WSDL (for the meta-information). This intermediate language (see Table 1) can notwithstanding be used (and, if necessary, expanded) to cover other orchestration languages.[3] A set of BPEL processes which form a service network are taken as the input to the analysis and the result is a logic program where BPEL processes are mapped onto predicates which call each other to mimic service invocations.

3.1 Overview of the Translation

The declarations in Table 1 can describe namespace prefixes, XML-schema-derived data types for messages, service port types, and external services that are not analyzed, but have some trusted properties (in this case, related to cost analysis) that are either given by a human or result from a separate analysis.

The activities supported by the intermediate language include generic constructs (`empty`, assignment, sequence,...) which are common to many programming languages as well as specific constructs to model orchestration workflows: `flow`, `float`, `scope/handler`, and `invoke`.

[3] Although, understandably, currently it explicitly deals with BPEL constructs.

Declarations and definitions	
Namespace prefix declaration	`:- prefix(` *Prefix*`,` *NamespaceURI*`).`
Message or complex type definition	`:- struct(` *QName*`,` *Members*`).`
Port type definition	`:- port_type(` *QName*`,` *Operations*`).`
External service declaration	`:- service(` *PortName*`,` *Operation*`, {` *Trusted properties* `}).`
Service definition	`service(` *Port*`,` *Operation*`,` *InMsg*`[,` *OutMsg*`]) :-` *Activity* `.`
Activities	
Do nothing	`empty`
Assignment to variable / part	*VarExpr* `<-` *Expr*
Service invocation	`invoke(` *PortName*`,` *Operation*`,` *OutMsg*`,` *InMsg*`)`
Terminating with a response	`reply(` *OutMsg*`)`
Sequence	$Activity_1$`,` $Activity_2$
Conditional execution	`if(` *Cond*`,` $Activity_1$`,` $Activity_2$`)`
While loop	`while(` *Cond*`,` *Activity*`)`
Repeat-until loop	`repeatUntil(` *Activity*`,` *Cond*`)`
For-each loop	`forEach(` *Counter*`,` *Start*`,` *End*`,` *Activity*`)`
Scope	`scope(` *VarDeclarations*`,` *Activities and Handlers*`)`
Scope fault handler	`handler(` *Activity*`)` `handler(` *FaultName*`,` *Activity*`)`
Parallel flow with dependencies	`flow(` *LinkDeclarations*`,` *Activities*`)`
Dependent activity in a flow	`float(` *Attributes*`,` *Activity*`)`

Table 1: Elements of an abstract description of an orchestration in the intermediate language.

In contrast to the structured workflow patterns expressed by UML activity/sequence diagrams, BPEL's `flow` construct can express a wider class of concurrent workflows, where concurrency and dependencies between activities are expressed by means of precondition formulas involving tri-state logical link variables, with optional dead-path elimination. The `float` construct in the intermediate language 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 BPEL process definition is translated into a service definition which associates a port name and an operation with a BPEL-style activity that represents the orchestration body. This intermediate representation is, in turn, translated into a logic programming language augmented with assertions (Ciao [10, 9]), which in our case are used to express types and modes (i.e., which arguments are input and output) as well as resource definitions and functions describing resource consumption bounds. The logic program resulting from the translation is fed to the resource consumption analyzer of the Ciao preprocessor (CiaoPP [9]), which is able to infer upper and lower bounds for the generalized cost / complexity of a logic program [6, 8, 14].

An important observation regarding the translation is that, in general, it is not necessary for the generated logic program to be strictly faithful to the operational semantics of the orchestration: it has to capture enough of it to ensure that the analyzers will infer correct information (i.e., safe approximations), with minimal precision loss due to the translation. However, in our case the translated program is executable (although not operationally equivalent to the BPEL process) and mirrors quite closely the operational semantics of the BPEL process under analysis.

```
:- 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 5: Translation of types.

3.2 Restrictions on Input Orchestrations

We restrict our analysis to orchestrations that follow a *receive–reply* interaction pattern, where processing activities take place after reception of an initiating message and finish dispatching either a reply or a fault notification. Orchestrations that may accept several different initiating messages can be logically decomposed into orchestrations that correspond to individual web service operations.

Another behavioral restriction is that we currently do not support analysis of stateful service callbacks using correlation sets or WS-Addressing schemes. In future work, we plan to relax both restrictions by identifying orchestration fragments that correspond to the *receive–reply* pattern, isolating them into sub-processes, and analyzing them as now done for whole orchestrations.

In our intermediate language, we support a variant of the `scope` construct, which, like its BPEL counterpart, introduces local variables, fault and compensation handlers. However, we do not fully support compensation handlers, which in BPEL contain logic that "undoes" effects of a successfully completed scope. The BPEL specification requires compensation handlers to use values of scope's variables that were recorded upon successful completion of the scope, which introduces problems for the analysis. Otherwise, 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

Services communicate using complex XML data structures whose typing information is given by an XML Schema. The state of an executing orchestration consists of a number of variables that have simple or complex types, including variables that hold inbound and outgoing messages. For simplicity reasons, we abstract the simple types in XML Schemata as three disjoint types: `numbers`, strings (represented by `atoms`), and `booleans`.

WSDL message types and custom complex types from XML Schemata are translated into the intermediate representation and finally into the typing / assertion language of Ciao. These type definitions are used to annotate the translated program and are eventually used by the analyzer. Figure 5 shows an automatically obtained translation for the part reservation scenario in Example 1. The type name `'factory->resData'` is a structure with the same name and 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).

We use a subset of XPath as the expression language, which allows node navigation only along the descendant and attribute axes, to ensure that navigation is statically decidable based on structural typing only. 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`. We also support a set of standard XPath operators and basic functions, including `position()` and `last()`.

To help the analyzer to track component values and correlate the changes made to them, we statically unfold XML structures in an environment into their components when necessary, and pass them around explicitly as predicate arguments from that point onwards. An unfolded structure no longer needs to be passed along with its components, since it can be reconstructed on demand (see Section 3.7 and Figure 3.7(c) for an example). The resulting code is less readable for a human, but more amenable to analysis.[4]

3.4 Basic Service and Activity Translation

The basic idea of the automatic translation from the intermediate language to a logic program is to keep track of the functional dependency between the message with which a service is invoked and the resulting response message. Thus, an orchestration S is translated into a predicate:

$$s(\bar{x}, y) \leftarrow [\![A]\!]_\eta(y)$$

where $\bar{x}$ represents the input message (decomposed in its parts), y stands for the answer, and $[\![A]\!]_\eta(y)$ is the translation of the orchestration body A within the initial service environment η. An environment is a mapping from structured component names within the current scope to logical terms. Structured component names denote parts within a message, nested XML nodes (elements and attributes), as well as heads and tails of lists. Each data structure is a tree of nodes rooted in a variable. Leaf nodes represent scalars and unfolded structured components. Since the internal nodes can be reconstructed from leaf nodes, the entire environment can be represented by its leaf nodes. Initially, the environment of an orchestration consists only of the input message (and its components). We write η in an argument position of a predicate to mean the leaf components from η. In the above case, we could have written $s(\eta, y)$ instead of $s(\bar{x}, y)$.

A sequence of activities $\langle A|C\rangle$ consists of the activity A and the continuation C (which is also a sequence of activities). A special case is the empty sequence ϵ. In general we consider the translation of a sequence, and abbreviate $[\![\langle A|C\rangle]\!]$ as $[\![A|C]\!]$, and $[\![A|\epsilon]\!]$ as $[\![A]\!]$. A sequence of two activities (A_i, A_j) is normalized by extending the continuation:

$$[\![(A_i, A_j)|C]\!]_\eta(y) \mapsto [\![A_i|\langle A_j|C\rangle]\!]_\eta(y)\,.$$

Activity `reply(`v`)` terminates the orchestration and sends a reply, regardless of the continuation. The translation produces a unification:

$$[\![\texttt{reply}(v)|C]\!]_\eta(y) \mapsto y = \eta(v)$$

between the service result y and the value of v in the current environment. Another way to terminate a service is to signal a fault, which is translated into a failure of the logical program:

$$[\![\texttt{throw}|C]\!]_\eta(y) \mapsto \texttt{fail}\,.$$

For any activity A_i other than a sequence, `empty`, `reply`, and `throw`, the translation is:

$$[\![A_i|C]\!]_\eta(y) \mapsto a_i(\eta, y)\,,$$

[4]The alternative being writing in Prolog the counterparts for the supported XPath operations and letting the analyzers deal directly with them. In our experience, this introduces too much precision loss in current analyzers, and therefore we opted for a more complex translation.

where a_i is a newly generated predicate whose structure depends on A_i, η, and C. First, we examine the case when $A_i \equiv x <- e$, i.e., the expression e is evaluated and assigned to the environment element x (a variable or its component). The generated clause consists of several steps:

$$a_i(\eta, y) \leftarrow [e : E]_\eta, [E/x]_\eta^{\eta'}, [\![C]\!]_{\eta'}(y).$$

Where $[e : E]_\eta$ stands for code that evaluates e into term E in environment η, and $[E/x]_\eta^{\eta'}$ stands for the assignment of E to x that transforms η into η'.

For an external service invocation, $A_i \equiv \texttt{invoke}(p, o, v, w)$, the generated clause has a similar structure:

$$a_i(\eta, y) \leftarrow s_{po}(\eta(v), Y), [Y/w]_\eta^{\eta'}, [\![C]\!]_{\eta'}(y),$$

where s_{po} is the translation of a service implementing operation o on port type p, v holds the input message, and w holds the reply.

For $A_i \equiv \texttt{if}(c, A_j, A_k)$, two clauses are generated:

$$a_i(\eta, y) \leftarrow [c?]_\eta, !, [\![A_j | C]\!]_\eta(y)$$
$$a_i(\eta, y) \leftarrow [\![A_k | C]\!]_\eta(y)$$

where $[c?]_\eta$ stands for the code that succeeds if and only if the boolean condition c evaluates to `true` in η. Likewise, $A_i \equiv \texttt{while}(c, A_j)$ generates:

$$a_i(\eta, y) \leftarrow [c?]_\eta, !, [\![A_j | \langle A_i \rangle]\!]_\eta(y)$$
$$a_i(\eta, y) \leftarrow [\![C]\!]_\eta(y)$$

Other looping constructs, such as `repeatUntil` and `forEach` reduce to `while`.

3.5 Translation for Scopes and Flows

The translation of scopes involves changing the environment on entry and exit, and has to ensure the execution of a fault handler unless the body scope ends successfully. In $A_i \equiv \texttt{scope}(D, A, H_1, H_2, \ldots, H_N)$, D denotes new variable declarations, A is the body of the scope, and H_i are fault handlers. $N+1$ clauses are generated for a_i, one for A and each of the handlers. Each of the clauses uses cut to prevent execution of subsequent clauses in case that the scope body / handler attached to the clause completes successfully. Since the process itself can be seen as a scope, and it normally needs a variable to hold the output message, in the intermediate language we use an abbreviation:

$$\texttt{service}(p, o, x, y) \leftarrow A$$

for:

$$\texttt{service}(p, o, x) \leftarrow \texttt{scope}([y : \mathit{ReplyType}], (A, \texttt{reply}(\texttt{'\$}y\texttt{'}))).$$

The translation of a `flow` is done following the usual BPEL semantics, but without operationally parallelizing the execution. Instead, we are interested in total resource consumption of a `flow` construct, irrespective of the actual number of available threads. Links are internally declared as Boolean variables, and `floats` are ordered so that they follow dependencies on outgoing links from previous `floats`. After reordering, a `flow` effectively translates to a sequence, and each $\texttt{float}(D, A_j)$ is transformed into:

$$\texttt{if}(c, (A_j, \texttt{'\$}o\texttt{' <- true}), F)$$

where c is a join condition, o is the outgoing link, and F covers the case when c evaluates to `false`. When the `suppresJoinFailure` property is disabled, we simply have $F \equiv \texttt{throw}(\texttt{bpel:joinFailure})$. Otherwise, $F \equiv \texttt{'\$}o\texttt{' <- false}$.

```
:- struct( factory:resRequest, [
part( body): struct( factory:resData)]).

:- struct( factory:resResponse, [
part( body): struct( factory: resData)]).

:- struct( factory:resData, [
child( factory:partCount): number,
child( factory:priceLimit): number,
child( factory:part):
  list( struct( factory:partInfo)) ]).

:- struct( factory:partInfo, [
attribute( '':partName): atom,
attribute( '':variantName): atom,
child( factory:partVendor): atom,
child( factory:serialNo): number ]).

:- port( factory:agent, [
reserveGroup( struct( factory:resRequest)):
   struct( factory:resResponse) ]).

:- port( factory:sales, [
reserveSingle( struct( factory:partInfo)):
       struct( factory:partInfo),
cancelReservation( struct( factory:partInfo)):
                      struct( factory:partInfo) ]).
```

```
service( factory:agent, reserveGroup, '$req', '$resp'):-
[
  '$resp.body/factory:partCount'<-0,
  '$resp.body/factory:part'<-'$req.body/factory:part',
  scope( [i:number],
  [   '$i' <- 1,
    while( '$req.body/factory:partCount>0',
    [
      scope( [p: struct( factory:partInfo),
      r: struct( factory:partInfo)],
      [   '$p'<- '$req.body/factory:part[$i]',
        invoke( factory:sales, reserveSingle, '$p', '$r'),
        if( '$r/factory:unitNo>0',
            '$resp.body/factory:part[$i]'<-'$r',
            throw( factory:unableToReserveGroup) ),
        handler(
        [   while( '$i>1',
          [   '$i'<- '$i - 1',
              '$p'<- '$resp.body/factory:part[$i]',
              invoke( factory:sales, cancelReservation,
                      '$p','$r')]),
              throw( factory:unableToCompleteRequest) ])
      ]),
      '$i' <- '$i+1',
      '$req.body/factory:partCount' <-
          '$req.body/factory:partCount - 1' ]) ]) ].
```

Figure 6: Abstract representation of a group reservation process.

3.6 Cost Functions for Closed-Source Services

During the analysis we have assumed that the orchestration code of the service(s) to be analyzed is available. However, this may not be always the case (see Section 2.3). The proposed solution was to publish the cost functions plus the size relations among arguments to be used by the analyzers. Another possibility is to make available the code representing the abstraction of the services so that it can be downloaded and directly used in the analysis. Hopefully, such code can be schematic enough not to reveal sensitive data about the actual service, but concrete enough to make inferring cost functions possible.

3.7 An Example of Translation and Analysis

We will illustrate the process of analysis by using a description of an orchestration, translating it into a logic program, and reasoning on the results of applying to it a resource usage analysis.

We use a representation of a process that performs part reservation, along the lines (but slightly simplified, for space reasons) of the example used in Section 2.1. For compactness, we present the abstract description of this orchestration in our internal representation form instead of plain BPEL (see Figure 6). This representation contains information that is both found in the WSDL document (data types, interface descriptions) and in the process definition itself (the processing logic).

```
:- entry 'service_factory->agent->reserveGroup'/4
  :{gnd,num}*{gnd,num}*{gnd,'list_of_factory->partInfo'}*var.

'service_factory->agent->reserveGroup'(A,B,C,D) :-
  act_1( A, B, C, 0, 0, [], D).
```

(a) Translation of the entry point to the process.

```
act_4( A, B, C, D, E, F, G, H):-
  ----(this is act_4:while('$req.body/factory:partCount>0')),
  A>0, !, act_5( A, B, C, D, E, F, G, H).
act_4( _, _, _, D, E, F, _, ', factory->resResponse'( D, E, F)).
```

(b) Translation of the main `while` loop.

```
act_7( A, B, C, D, E, F, G, H, _, _, _, _, M):-
  ----(this is act_7:invoke( factory:sales, reserveSingle, '$p', '$r')),
  H='factory->partInfo'(N, O, P, Q),
  'service_factory->sales->reserveSingle'( N, O, P, Q, R),
  act_8( A, B, C, D, E, F, G, N, O, P, Q, R, M).
```

(c) Translation of an external service invocation.

Figure 7: Translation into a logic program.

The orchestration traverses the list of parts to reserve, external factory sales service.[5] If that is not possible, or if a failure arises, a failure handler is activated that tries to cancel the reservations that were already made before signaling failure to the client.

The translation of the orchestration produces an annotated logic program, some of whose parts we present in Figure 3.7. Part (a) shows the translation of the entry point of the service, along with an `entry` annotation that helps the analyzer understand what the input arguments are. The input message is unfolded into the first three arguments (A, B, C), and D plays the role of y. Part (b) shows the translation of the main `while` loop, and the second clause finishes the process by constructing the answer from the current value of the response variable. Part (c) shows the translation of the service invocation, with previous unfolding of the outgoing message, and subsequent pruning of the response variable data tree.

The resource analysis finds out how many times some specific operations will be called during the execution of the process. The resources we are interested in in this example are: the number of all basic activities performed (assignments, external invocations); the number of invocations of individual part reservations (operation `reserveSingle` at the factory service); and the number of invocations of reservation cancellations (operation `cancelReservation` at the factory service). From the number of invocations it is easy to deduce the number of messages exchanged during the execution of the process. The results are displayed in Table 2, where the estimated upper and lower bounds are expressed as a function of the initiating request. We differentiate explicitly two cases: one which has the possibility of failure, in which the associated fault handling is executed, which gives wider, more cautious estimates, and another one in which the execution is successful (i.e., without fault generation and handling). These two cases were obtained by means of different translations which explicitly generated or not Prolog code corresponding to the fault handling.

[5] This is a difference from Example 1: the orchestration does not query different factories.

Resource	With fault handling		Without fault handling	
	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

Note: In the above formula, n stands for the value of the input argument `$req.body/factory:partCount`, taken as a non-negative integer.

Table 2: Resource analysis results for the group reservation service.

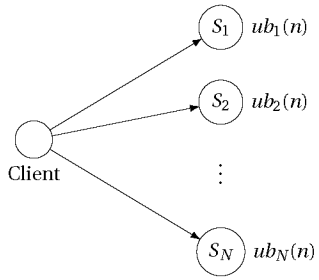


Figure 8: Single-tier simulation setting.

4 An Experiment on Adaptation

We will study the effectiveness of applying our data-aware cost analysis to adaptation by simulating several arrangements of service networks. We consider adaptation by means of dynamic (re-)selection of partner services using several strategies (see later for more details). The goal is to assess their relative efficiency (in terms of benefits) when considering worst-case execution scenarios, where service executions incur maximal costs under their stated upper bounds.

In the simulations, we consider the cost in terms of number of messages exchanged. That is meaningful in situations where the number of messages reflects the intensity of computation within the service network. However, as we argued before, the concrete meaning of the cost functions is not really relevant, and as long as they safely measure a characteristic which we want to maximize (minimize), the techniques and experiments herein presented can be applied to other characteristics.

We also make the realistic assumption that the selected service may be unavailable or fail to fulfill the expected task within the limits set by the user or mandated by some Service-Level Agreement. In both cases, we continue with the selection and invocation of the "next-best" candidate from the partner pool. For the sake of simplicity, and although in reality failure probabilities vary from one service to another, in each simulation we adopt a single partner failure probability p_f.

Within the general setting of the car part reservation system (Section 2.1), we simulate the behavior of two arrangements of services. The first arrangement, shown in Figure 8, is a single tier of services that provide the reservation of particular car parts. There is a pool of $N = 12$ part provider services with different resource consumption features. The client invokes one of these services to reserve n units of a particular part type. Upper bounds for services in the pool are shown on Figure 9. For the input range of 0 to 50 requested units, the upper bounds are a family of curves that were chosen to maximize data-aware choice opportunities. The upper bound of the first service (marked as $ub_1(n)$

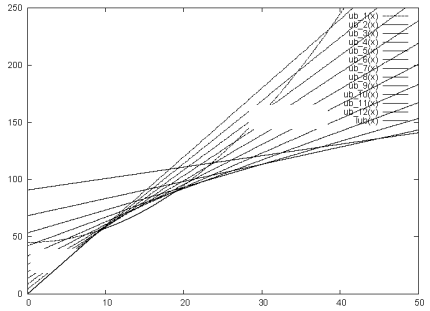


Figure 9: Cost upper bounds for different services, single-tier setting.

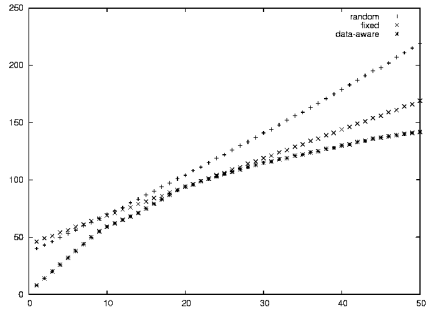


Figure 10: Single-tier simulation results for $p_f = 0$.

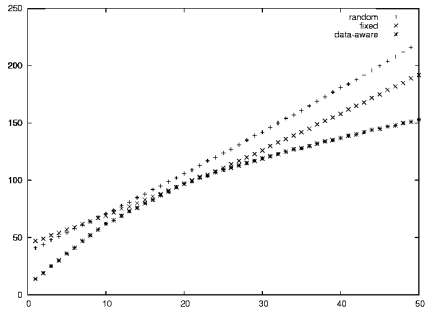


Figure 11: Single-tier simulation results for $p_f = 0.5$.

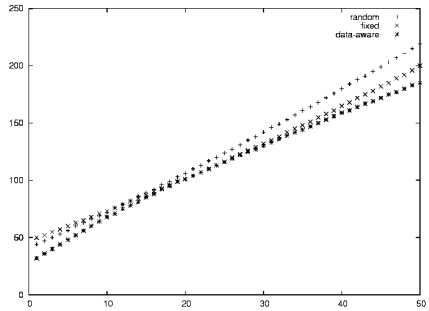


Figure 12: Single-tier simulation results for $p_f = 0.8$.

on Figure 8) is quadratic on the input size n, and has the form:

$$ub_1(n) = An^2 + B,$$

Other services from the pool have linear upper bounds of the form:

$$ub_k(n) = k \cdot \alpha \cdot n + \beta \sum_{i=k}^{N-1} \frac{1}{i},$$

($k = 2..N$) with choice of $\alpha = 1/2$ and $\beta = 45$. The underlying bold black line on Figure 9, marked with lub, is the least upper bound for each given n in the input range — i.e., it describes the best possible case among the more pessimistic prediction for all the available services and for each n in the data range

The simulation uses three selection strategies: (a) a *random choice* of service; (b) a *data-aware cost-minimizing choice* based on the input data, which selects service offering the least upper bound for a given n; and (c) *fixed preferences* over services, where the cost associated to every service is a constant corresponding to its actual cost for the input size $n = 20$. Each result obtained from the simulation is the average from one hundred simulations of service invocations.

The simulation results for the single-tier setting with $p_f = 0$ (i.e., without failure) are shown in Figure 10. Unsurprisingly, the costs using the data-aware selection strategy closely follow the least

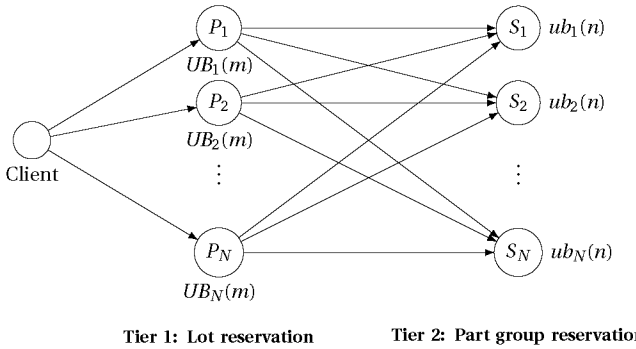


Figure 13: Two-tier simulation setting.

upper bound curve from Figure 9 and, naturally, it meets the *fixed preferences* strategy at $n = 20$. Moving to the left or to the right the point chosen to create the fixed preferences would of course make this strategy loose precision on the other end.

In terms of cost savings, the data-aware strategy is significantly better than random choice, and also beats fixed preferences. The simulation also shows that the advantage of the data-aware strategy is resilient to increments of the fault rate. In Figure 11, with $p_f = 0.5$, which is a very high fault rate, the data-aware strategy on average maintains its benefits, which are close to be lost only at very high failure rates, such as $p_f = 0.8$ (Figure 12). This experimental data supports the claim that taking into account actual data when performing service selection bring substantial gains.

The second simulation arrangement, shown on Figure 13, consists of two tiers of services. The first tier, invoked by the client, consists of services that reserve a mix of $M = 5$ different types of parts for a lot of m vehicles. Each vehicle requires c_i units of part type i, $i = 1..M$. These parts are obtained from the second tier of part providers, which behave in the same way as in the single-tier case. The upper bound cost (in terms of messages exchanged) $UB_j(m)$ of a lot reservation service j in the first tier is given as:

$$UB_j(m) = E_j(m) + M + \sum_{i=1}^{M} ub_*(m \cdot c_i)$$

where $E_j(m)$ stands for the upper bound of the internal, structural cost of j, M is added for each invocation of a part reservation service, and ub_* stands for the upper bound of the cost of reserving $m \cdot c_i$ parts of type i. Under a data-aware selection strategy, ub_* corresponds to the second-tier service with lowest upper bound for the given $n = m \cdot c_i$ that is selected by the first-tier service j. In the experiment we took the same number of $N = 12$ second- and first-tier services, and varied their structural complexity to have both the quadratic case:

$$E_1(m) = Cm^2 + D,$$

and linear cases:

$$E_j(m) = j \cdot \gamma \cdot m + \delta \sum_{i=j}^{N-1} \frac{1}{i},$$

($j = 2..N$) with $\gamma = 5/2$ and $\delta = 225$. Thus, the relationship between different E_j ($j = 1..N$) is analogous to the relationship between different ub_k ($k = 1..N$) in Figure 9. The meaning of E_j is the number of messages exchanged between the first-tier service j and entities that are either passive (e.g., filing repositories or mail message recipients) or have a constant upper-bound cost function, that does not depend on a particular m.

In the two-tier case we also have three selection strategies. *Random selection*, which applies both to invocations from the client to the first tier services, and to invocations from the first to the second service layer. *Data-aware selection* works in a more sophisticated manner, in order to account for costs incurred by services in both layers. In this particular simulation, we have taken the *top-down* approach, where first-tier services are queried for their total upper bound cost, including the costs of the second-tier services they invoke. In order to present their total upper bound cost, relative to a particular input value of m, the first-tier services perform pre-selection, i.e., advance planning of second-tier partner links, to which they stick if selected by the client.

Although in our case the top-down pre-selection process ends with the second-tier services, in reality it can extend until it either reaches the terminal points (atomic services that do the actual work), or detects a circular reference between services, in which case the tier-to-tier cost dependencies effectively turn into a set of recurrent relations that need to be solved using adequate mathematical methods. However, we argue that the existence of such circularities is not common, since service networks usually rely on back-ends of "worker" services that are atomic in the sense that they do not rely on the invocation of other external services.

Another approach to select services based on data-related functions would be to approximate the cost bounds of a collection of connected services with the corresponding structural costs, which do not depend on bindings the orchestration may make. While we have explored this possibility, for space reasons we are not reporting. It amounts to saying that the structural cost is not necessarily a good predictor of the actual costs of a composition after partner binding and that, in any case, it cannot be used as real upper bound as it does not provide a guarantee for the real cost function.

The third partner selection strategy in the two-tier setting is again *fixed preferences* over services in both tiers. This time, we form the preference over services in the first tier by minimizing their structural costs for the particular $m = 20$.

The results of the simulation for the two-tier setting and $p_f = 0$ are shown in Figure 14. We again notice that the data-aware top-down approach with pre-planning beats (by far this time) both the random and fixed-preferences strategies. Again, the results are resilient to increments of the fault rate, and tend to deteriorate only at very high failure rates (Figures 15 and 16) which, again, gives a strong support to the use of cost functions in the cases where they can be applied.

5 Conclusions and Future Work

We have presented a resource analysis for orchestrations (using BPEL as a concrete example) which is based on a translation into Prolog, for which cost analyzers are available, via an intermediate programming language. These analyzers can be customized to focus on user-defined resources, thereby opening the possibility of generating cost functions for characteristics other than computation complexity, some of them relevant for SOC. As we argued and showed by simulations, automatically inferring and applying these functions can be used as a core technology for some approaches to service adaptation and matchmaking.

We sketched the core of the translation process, which approximates the behavior of the original process network in such a way that the analysis results are valid for the original network. Our translation is partial in the sense that some issues, like correlation sets, are not yet taken into account. A richer translation which we expect will take into account this (and other) issues is the subject of current work.

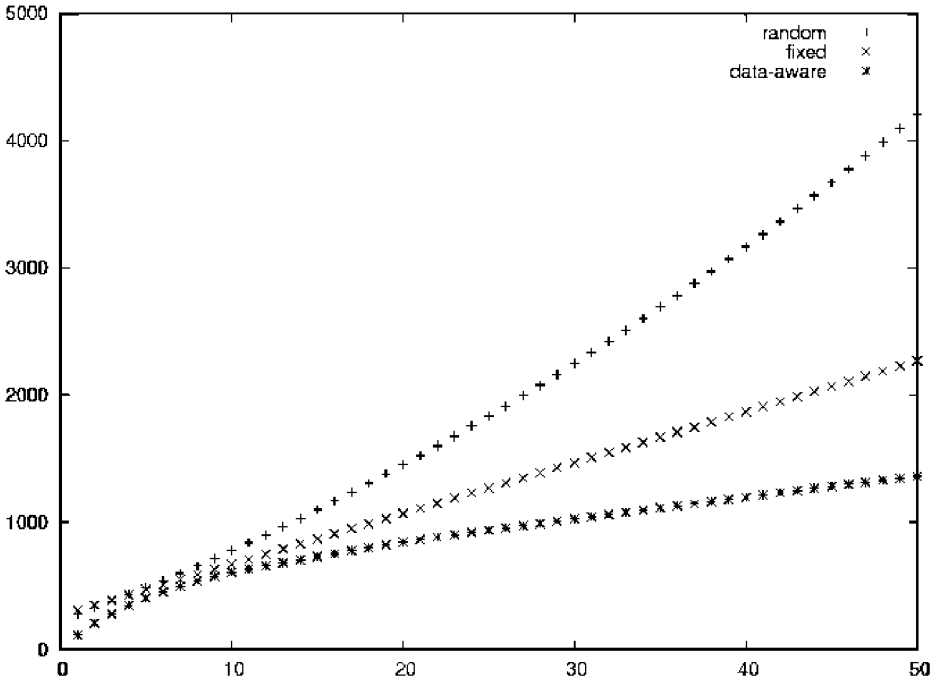


Figure 14: Two-tier simulation results for $p_f = 0$.

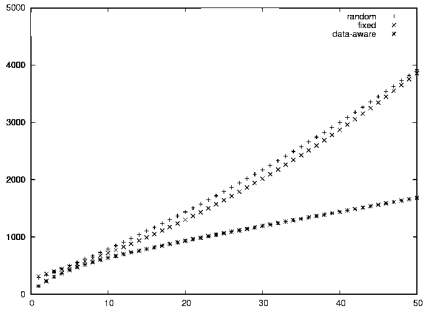


Figure 15: Two-tier simulation results for $p_f = 0.5$.

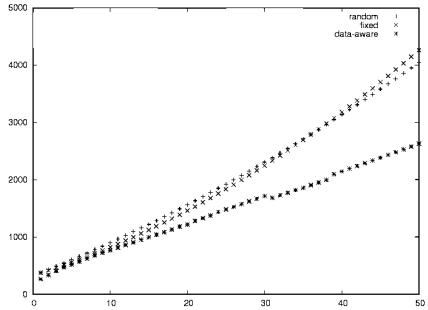


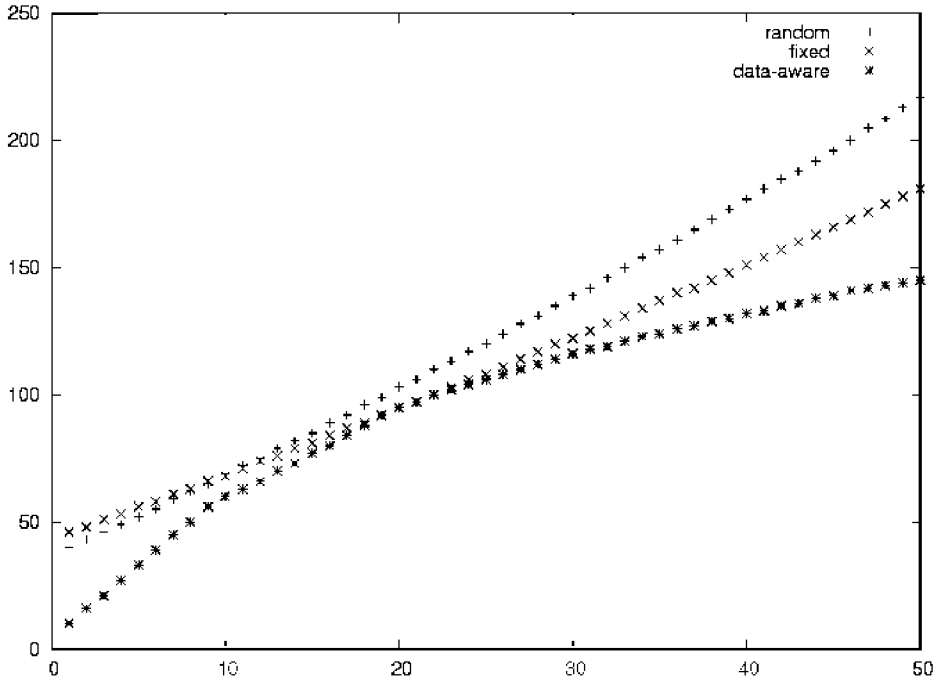
Figure 16: Two-tier simulation results for $p_f = 0.8$.

Finally, we performed a series of experiments with different adaptation strategies and services which support the usefulness of using data functions in the selection of services.

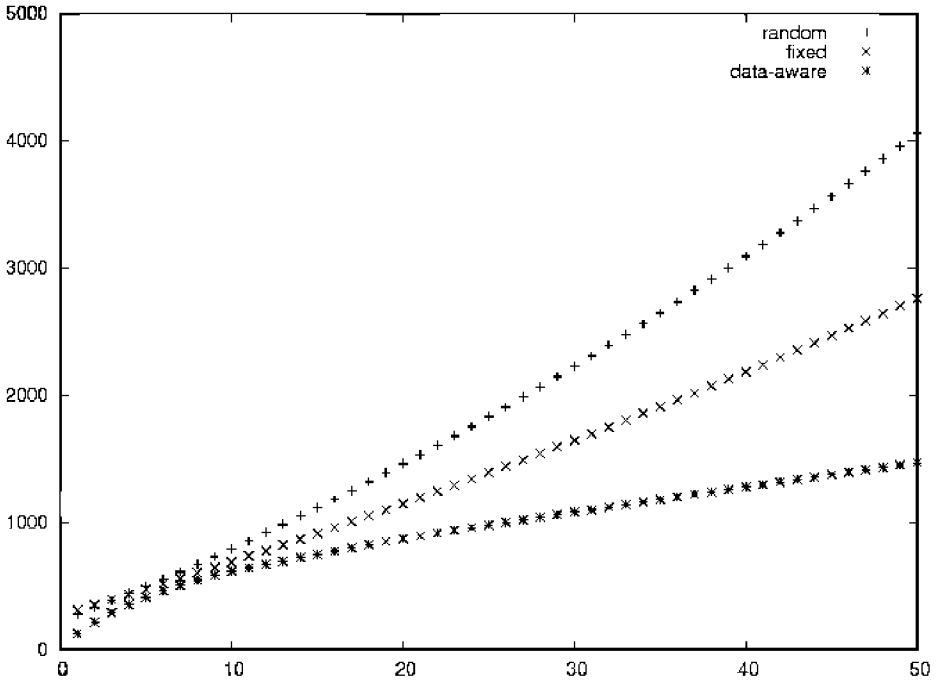
A Raw data from experiments

In order for the reviewers to have all the information coming from the experiments (which we depicted in Figures 10 to 12), we are including here tables for all the data points we generated. Note that in the figures we are not including the plot for the case with failure 0.25 (shown in Figure 3) as it is not very different from that for failure rate 0.5 (shown in Figure 11).

The simulation results in Figure 3 show resilience of the data-aware selection strategy in the single-tier setting, when a significant fraction (one quarter) of service calls fails. Compared to Figure 10 with $p_f = 0$, the shape of cost curves does not significantly change, except that they are slightly shifted upwards to account for repeated calls to second-best, third-best, etc. service in a row. The same applies to Figure 11 in comparison to 14.



Single-tier simulation results with $p_f = 0.25$.



Two-tier simulation results with $p_f = 0.25$.

	$p_f = 0.0$			$p_f = 0.25$			$p_f = 0.5$			$p_f = 0.8$		
n	rnd.	fixed	u.b.	rnd.	fixed	u.b.	rnd.	fixed	u.b.	rnd.	fixed	u.b.
1	40	46	8	40	46	10	41	47	14	44	50	32
2	43	49	14	43	48	16	44	49	19	47	52	36
3	46	51	20	46	51	21	48	52	25	50	55	40
4	50	54	26	49	53	27	51	54	30	53	57	44
5	53	56	32	52	56	33	54	57	36	56	60	48
6	56	59	38	55	58	39	58	59	41	60	63	52
7	60	61	44	59	61	45	61	62	47	63	65	56
8	63	64	50	62	63	50	64	64	52	66	68	60
9	66	66	55	65	66	56	68	67	57	69	71	64
10	70	69	59	69	68	60	71	69	62	72	73	68
11	73	71	62	72	71	63	74	72	65	76	76	71
12	76	74	65	75	74	66	78	75	69	79	79	75
13	80	76	68	79	76	70	81	77	73	82	82	78
14	83	79	71	82	79	73	85	80	76	86	84	81
15	87	81	75	85	81	77	88	83	80	89	87	85
16	90	84	79	89	84	80	92	86	83	92	90	88
17	94	86	83	92	87	84	95	88	87	96	93	92
18	97	89	87	96	89	88	99	91	90	99	96	95
19	101	91	91	99	92	92	102	94	94	103	99	99
20	104	94	94	103	95	95	106	97	97	106	101	101
21	108	96	96	106	97	97	109	100	99	110	104	104
22	111	99	99	110	100	100	113	103	102	113	107	107
23	115	101	101	113	103	102	117	105	104	117	110	110
24	119	104	103	117	106	104	120	108	107	120	113	113
25	122	106	105	120	108	106	124	111	109	124	116	116
26	126	109	107	124	111	108	127	114	111	127	120	119
27	130	111	109	128	114	110	131	117	113	131	123	122
28	133	114	111	131	117	112	135	120	115	135	126	125
29	137	116	113	135	120	114	139	123	117	138	129	127
30	141	119	115	139	122	116	142	126	119	142	132	130
31	144	121	116	142	125	118	146	130	121	146	135	133
32	148	124	118	146	128	119	150	133	123	149	138	136
33	152	126	119	150	131	121	154	136	125	153	142	139
34	156	129	121	154	134	123	157	139	127	157	145	142
35	159	131	122	157	137	124	161	142	128	161	148	144
36	163	134	124	161	140	126	165	145	130	164	151	147
37	167	136	125	165	142	127	169	149	132	168	155	150
38	171	139	127	169	145	129	173	152	134	172	158	153
39	175	141	128	173	148	130	177	155	135	176	161	156
40	179	144	130	177	151	132	181	158	137	180	165	159
41	183	146	131	181	154	133	184	162	139	184	168	161
42	187	149	133	185	157	135	188	165	140	188	172	164
43	191	151	134	188	160	136	192	168	142	191	175	167
44	195	154	136	192	163	138	196	172	144	195	178	169
45	198	156	137	196	166	139	200	175	145	199	182	172
46	202	159	138	200	169	141	204	179	147	203	185	175
47	207	161	139	205	172	142	208	182	148	207	189	177
48	211	164	140	209	175	143	212	185	150	211	192	180
49	215	166	141	213	178	144	216	189	151	215	196	183
50	219	169	142	217	181	145	221	192	153	219	200	185

Table 3: The simulation data for the single-tier setting.

	$p_f = 0.0$			$p_f = 0.25$			$p_f = 0.5$			$p_f = 0.8$		
n	rnd.	fixed	u.b.	rnd.	fixed	u.b.	rnd.	fixed	u.b.	rnd.	fixed	u.b.
1	276	309	113	277	308	123	285	316	142	377	374	262
2	327	349	206	330	349	214	337	356	229	432	418	335
3	379	389	278	384	390	287	391	397	303	488	463	404
4	432	429	345	440	431	353	445	439	367	544	510	469
5	487	469	403	496	473	410	500	483	424	602	558	517
6	542	509	452	553	515	459	556	527	473	660	608	571
7	599	549	495	611	557	503	613	574	518	719	659	623
8	657	589	536	670	600	543	671	622	560	779	712	678
9	717	629	575	731	643	582	730	671	600	840	766	725
10	777	669	606	792	687	615	790	721	636	901	822	768
11	839	709	632	854	731	642	850	773	668	964	879	814
12	902	749	656	917	776	669	912	826	699	1027	938	859
13	967	789	680	981	820	695	974	881	730	1091	998	905
14	1032	829	704	1047	866	721	1037	937	760	1156	1059	950
15	1099	869	729	1113	911	747	1102	994	790	1221	1122	995
16	1167	909	752	1180	958	772	1167	1053	819	1288	1187	1040
17	1236	949	775	1248	1004	797	1233	1113	848	1355	1253	1085
18	1307	989	799	1317	1051	823	1300	1174	878	1424	1321	1130
19	1378	1029	822	1388	1098	848	1367	1237	907	1493	1390	1182
20	1451	1069	842	1459	1146	870	1436	1301	934	1563	1460	1215
21	1525	1109	862	1531	1194	892	1506	1366	961	1633	1532	1283
22	1601	1149	882	1604	1242	914	1576	1433	988	1705	1606	1328
23	1677	1189	901	1678	1291	936	1648	1502	1014	1777	1680	1374
24	1755	1229	919	1754	1340	956	1720	1571	1040	1851	1757	1435
25	1834	1269	937	1830	1390	977	1793	1642	1066	1925	1835	1481
26	1914	1309	955	1907	1440	998	1868	1715	1091	2000	1914	1526
27	1996	1349	973	1985	1491	1018	1943	1788	1117	2075	1995	1576
28	2079	1389	991	2064	1542	1039	2019	1863	1142	2152	2077	1622
29	2163	1429	1009	2145	1593	1059	2096	1940	1167	2229	2161	1669
30	2248	1469	1027	2226	1644	1080	2173	2018	1193	2307	2246	1715
31	2334	1509	1044	2308	1697	1099	2252	2097	1218	2386	2333	1684
32	2422	1549	1061	2391	1749	1119	2332	2178	1243	2466	2421	1728
33	2511	1589	1078	2475	1802	1139	2412	2260	1267	2547	2511	1773
34	2601	1629	1095	2561	1855	1159	2494	2343	1292	2629	2602	1818
35	2692	1669	1112	2647	1909	1178	2576	2428	1317	2711	2695	1860
36	2785	1709	1129	2734	1963	1198	2659	2514	1342	2794	2789	1906
37	2879	1749	1146	2822	2017	1218	2743	2601	1367	2878	2885	1952
38	2974	1789	1163	2911	2072	1237	2828	2690	1391	2963	2982	1998
39	3070	1829	1180	3001	2128	1257	2914	2780	1416	3049	3081	2098
40	3168	1869	1197	3093	2183	1276	3001	2872	1440	3135	3181	2145
41	3266	1909	1214	3185	2239	1296	3089	2965	1465	3223	3282	2192
42	3366	1949	1231	3278	2296	1315	3177	3059	1490	3311	3385	2240
43	3468	1989	1248	3372	2353	1335	3267	3154	1514	3400	3490	2288
44	3570	2029	1265	3467	2410	1354	3357	3252	1539	3490	3596	2336
45	3674	2069	1282	3564	2468	1374	3449	3350	1563	3581	3703	2384
46	3779	2109	1298	3661	2526	1393	3541	3450	1587	3672	3812	2433
47	3885	2149	1314	3759	2584	1411	3634	3551	1612	3765	3923	2481
48	3992	2189	1330	3858	2643	1430	3728	3653	1636	3858	4035	2530
49	4101	2229	1346	3958	2702	1449	3823	3757	1660	3952	4148	2579
50	4210	2269	1362	4060	2762	1468	3919	3862	1684	4047	4263	2628

Table 4: The simulation data for the two-tier setting.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Relation Systems: a Language–Independent Target Language for Cost Analysis. In *Spanish Conference on Programming and Computer Languages (PROLE'08)*, volume 17615 of *ENTCS*. Elsevier, October 2008.

2. E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Ramírez, and D. Zanardini. The COSTA cost and termination analyzer for java bytecode and its web interface (tool demo). In Anna Philippou, editor, *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, July 2008.

3. 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.

4. J. Cardoso. About the Data-Flow Complexity of Web Processes. In *6th International Workshop on Business Process Modeling, Development, and Support: Business Processes and Support Systems: Design for Flexibility*, pages 67–74, 2005.

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

6. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

7. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

8. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

9. 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.

10. 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 Jose Meseguer Pierpaolo Degano, Rocco De Nicola, editor, *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

11. D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. Kevin Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, BEA, Intalio, Individual, Adobe Systems, Systinet, Active Endpoints, JBoss, Sterling Commerce, SAP, Deloitte, TIBCO Software, webMethods, Oracle, 2007.

12. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

13. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2009.

14. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

15. Á. Rebón Portillo, K. Hammond, H-W. Loidl, and P. Vasconcelos. Cost Analysis Using Automatic Size and Time Inference. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 232–247, Madrid, Spain, September 2002. Springer-Verlag.

16. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem – Overview of Methods And Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(36), 2008.