Towards Data-Aware Cost-Driven Adaptation for Service Orchestrations

facultad de informática
universidad politecnica 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.
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.
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, 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

---

2Note, however, that when the inferred upper and lower bounds coincide they are exact cost functions.
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 2: Upper and lower bounds for two services.
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

\[
T_A(n) = 2n + 3 + n(n + 5) = n^2 + 7n + 3 \quad \text{for } B_1
\]
\[
T_A(n) = 2n + 3 + n(2n + 1) = 2n^2 + 4n + 3 \quad \text{for } B_2
\]

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
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.

3Although, 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 Activity₁, Activity₂
- Conditional execution if(Cond, Activity₁, Activity₂)
- 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 pre-condition 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.
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 stat-
ically 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. 

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:

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

where $\dot{x}$ 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 $\eta$. An environ-
ment 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 at-
tributes), 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 $\eta$ in an argument position of a predicate to mean the leaf components from $\eta$. In the above
case, we could have written $s(\eta, y)$ instead of $s(\dot{x}, y)$.

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 $\epsilon$. In general we consider the translation
of a sequence, and abbreviate $[[A|C]]$ as $[[A|C]]$, and $[A|\epsilon]$ as $[A]$. A sequence of two activities $(A_i, A_j)$
is normalized by extending the continuation:

$$ [(A_i, A_j)|C]_\eta(y) \leftarrow [A_i](A_j|C)_\eta(y) $$

Activity $\text{reply}(v)$ terminates the orchestration and sends a reply, regardless of the continuation.
The translation produces a unification:

$$ [[\text{reply}(v)|C]]_\eta(y) \leftarrow y = \eta(v) $$

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:

$$ [[\text{throw}|C]]_\eta(y) \leftarrow \text{fail} $$

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

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

$^4$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 \( A_i, \eta \), and \( C \). First, we examine the case when \( A_i = x \leftarrow e \), 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 \( \eta \), and \([E/x]_\eta^\eta\) stands for the assignment of \( E \) to \( x \) that transforms \( \eta \) into \( \eta' \).

For an external service invocation, \( A_i = \text{invoke}(p, o, v, w) \), 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 \( A_i = \text{if}(c, A_j, A_k) \), two clauses are generated:

\[
a_i(\eta, y) \leftarrow [c?]_\eta^{\eta}, [A_j]_\eta^{\eta}(y)
\]

\[
a_i(\eta, y) \leftarrow [A_k]_\eta(y)
\]

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

\[
a_i(\eta, y) \leftarrow [c?]_\eta^{\eta}, [A_j]_\eta^{\eta}(y)
\]

\[
a_i(\eta, y) \leftarrow [C]_\eta^{\eta}(y)
\]

Other looping constructs, such as \text{repeatUntil} and \text{forEach} reduce to \text{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 \( A_i = \text{scope}(D, A, H_1, H_2,...,H_N) \), \( D \) denotes new variable declarations, \( A \) is the body of the scope, and \( H_j \) 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:

\[
\text{service}(p, o, x, y) \leftarrow A
\]

for:

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

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

\[
\text{if}(c, (A_j, '\$o' \leftarrow \text{true}), F)
\]

where \( c \) is a join condition, \( o \) is the outgoing link, and \( F \) covers the case when \( c \) evaluates to \( \text{false} \). When the \text{suppressJoinFailure} property is disabled, we simply have \( F \equiv \text{throw(bpel:joinFailure)} \). Otherwise, \( F \equiv '\$o' \leftarrow \text{false} \).
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).
The orchestration traverses the list of parts to reserve, external factory sales service. 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 \(A, B, C, D\), 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.
<table>
<thead>
<tr>
<th>Resource</th>
<th>With fault handling</th>
<th>Without fault handling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lower bound</td>
<td>upper bound</td>
</tr>
<tr>
<td>Basic activities</td>
<td>2</td>
<td>$7 \times n$</td>
</tr>
<tr>
<td>Single reservations</td>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>Cancellations</td>
<td>0</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>

**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 $pf$.

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
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:

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

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 $m$. 

Figure 13: Two-tier simulation setting.
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.
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 \( p_f = 0.25 \).](image-url)
Two-tier simulation results with $p_f = 0.25$. 
<table>
<thead>
<tr>
<th>$n$</th>
<th>$p_f = 0.0$</th>
<th>$p_f = 0.25$</th>
<th>$p_f = 0.5$</th>
<th>$p_f = 0.8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rnd.</td>
<td>fixed</td>
<td>u.b.</td>
<td>rnd.</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>46</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
<td>49</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>51</td>
<td>20</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>54</td>
<td>26</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>56</td>
<td>32</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>56</td>
<td>59</td>
<td>38</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
<td>61</td>
<td>44</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>63</td>
<td>64</td>
<td>50</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>66</td>
<td>66</td>
<td>55</td>
<td>65</td>
</tr>
<tr>
<td>10</td>
<td>70</td>
<td>69</td>
<td>59</td>
<td>69</td>
</tr>
<tr>
<td>11</td>
<td>73</td>
<td>71</td>
<td>62</td>
<td>72</td>
</tr>
<tr>
<td>12</td>
<td>76</td>
<td>74</td>
<td>65</td>
<td>75</td>
</tr>
<tr>
<td>13</td>
<td>80</td>
<td>76</td>
<td>68</td>
<td>79</td>
</tr>
<tr>
<td>14</td>
<td>83</td>
<td>79</td>
<td>71</td>
<td>82</td>
</tr>
<tr>
<td>15</td>
<td>87</td>
<td>81</td>
<td>75</td>
<td>85</td>
</tr>
<tr>
<td>16</td>
<td>90</td>
<td>84</td>
<td>79</td>
<td>89</td>
</tr>
<tr>
<td>17</td>
<td>94</td>
<td>86</td>
<td>83</td>
<td>92</td>
</tr>
<tr>
<td>18</td>
<td>97</td>
<td>89</td>
<td>87</td>
<td>96</td>
</tr>
<tr>
<td>19</td>
<td>101</td>
<td>91</td>
<td>91</td>
<td>99</td>
</tr>
<tr>
<td>20</td>
<td>104</td>
<td>94</td>
<td>94</td>
<td>103</td>
</tr>
<tr>
<td>21</td>
<td>108</td>
<td>96</td>
<td>96</td>
<td>106</td>
</tr>
<tr>
<td>22</td>
<td>111</td>
<td>99</td>
<td>99</td>
<td>110</td>
</tr>
<tr>
<td>23</td>
<td>115</td>
<td>101</td>
<td>101</td>
<td>113</td>
</tr>
<tr>
<td>24</td>
<td>119</td>
<td>104</td>
<td>103</td>
<td>117</td>
</tr>
<tr>
<td>25</td>
<td>122</td>
<td>106</td>
<td>105</td>
<td>120</td>
</tr>
<tr>
<td>26</td>
<td>126</td>
<td>109</td>
<td>107</td>
<td>124</td>
</tr>
<tr>
<td>27</td>
<td>130</td>
<td>111</td>
<td>109</td>
<td>128</td>
</tr>
<tr>
<td>28</td>
<td>133</td>
<td>114</td>
<td>111</td>
<td>131</td>
</tr>
<tr>
<td>29</td>
<td>137</td>
<td>116</td>
<td>113</td>
<td>135</td>
</tr>
<tr>
<td>30</td>
<td>141</td>
<td>119</td>
<td>115</td>
<td>139</td>
</tr>
<tr>
<td>31</td>
<td>144</td>
<td>121</td>
<td>116</td>
<td>142</td>
</tr>
<tr>
<td>32</td>
<td>148</td>
<td>124</td>
<td>118</td>
<td>146</td>
</tr>
<tr>
<td>33</td>
<td>152</td>
<td>126</td>
<td>119</td>
<td>150</td>
</tr>
<tr>
<td>34</td>
<td>156</td>
<td>129</td>
<td>121</td>
<td>154</td>
</tr>
<tr>
<td>35</td>
<td>159</td>
<td>131</td>
<td>122</td>
<td>157</td>
</tr>
<tr>
<td>36</td>
<td>163</td>
<td>134</td>
<td>124</td>
<td>161</td>
</tr>
<tr>
<td>37</td>
<td>167</td>
<td>136</td>
<td>125</td>
<td>165</td>
</tr>
<tr>
<td>38</td>
<td>171</td>
<td>139</td>
<td>127</td>
<td>169</td>
</tr>
<tr>
<td>39</td>
<td>175</td>
<td>141</td>
<td>128</td>
<td>173</td>
</tr>
<tr>
<td>40</td>
<td>179</td>
<td>144</td>
<td>130</td>
<td>177</td>
</tr>
<tr>
<td>41</td>
<td>183</td>
<td>146</td>
<td>131</td>
<td>181</td>
</tr>
<tr>
<td>42</td>
<td>187</td>
<td>149</td>
<td>133</td>
<td>185</td>
</tr>
<tr>
<td>43</td>
<td>191</td>
<td>151</td>
<td>134</td>
<td>188</td>
</tr>
<tr>
<td>44</td>
<td>195</td>
<td>154</td>
<td>136</td>
<td>192</td>
</tr>
<tr>
<td>45</td>
<td>198</td>
<td>156</td>
<td>137</td>
<td>196</td>
</tr>
<tr>
<td>46</td>
<td>202</td>
<td>159</td>
<td>138</td>
<td>200</td>
</tr>
<tr>
<td>47</td>
<td>207</td>
<td>161</td>
<td>139</td>
<td>205</td>
</tr>
<tr>
<td>48</td>
<td>211</td>
<td>164</td>
<td>140</td>
<td>209</td>
</tr>
<tr>
<td>49</td>
<td>215</td>
<td>166</td>
<td>141</td>
<td>213</td>
</tr>
<tr>
<td>50</td>
<td>219</td>
<td>169</td>
<td>142</td>
<td>217</td>
</tr>
</tbody>
</table>

Table 3: The simulation data for the single-tier setting.
<table>
<thead>
<tr>
<th>$n$</th>
<th>$p_f = 0.0$</th>
<th>$p_f = 0.25$</th>
<th>$p_f = 0.5$</th>
<th>$p_f = 0.8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>276</td>
<td>309</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>327</td>
<td>349</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>379</td>
<td>389</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>432</td>
<td>429</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>487</td>
<td>469</td>
<td>403</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>542</td>
<td>509</td>
<td>452</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>599</td>
<td>549</td>
<td>485</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>657</td>
<td>589</td>
<td>556</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>717</td>
<td>629</td>
<td>575</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>777</td>
<td>669</td>
<td>606</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>839</td>
<td>709</td>
<td>632</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>902</td>
<td>749</td>
<td>656</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>967</td>
<td>786</td>
<td>688</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1032</td>
<td>829</td>
<td>764</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1099</td>
<td>869</td>
<td>729</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1167</td>
<td>908</td>
<td>752</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1230</td>
<td>949</td>
<td>775</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1307</td>
<td>989</td>
<td>799</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1376</td>
<td>1029</td>
<td>822</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1451</td>
<td>1069</td>
<td>842</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1525</td>
<td>1109</td>
<td>862</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>1601</td>
<td>1149</td>
<td>882</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1677</td>
<td>1189</td>
<td>901</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1755</td>
<td>1229</td>
<td>919</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1834</td>
<td>1269</td>
<td>937</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1914</td>
<td>1309</td>
<td>955</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1996</td>
<td>1349</td>
<td>973</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>2079</td>
<td>1389</td>
<td>991</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>2163</td>
<td>1429</td>
<td>1009</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2248</td>
<td>1469</td>
<td>1027</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>2334</td>
<td>1509</td>
<td>1044</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2422</td>
<td>1549</td>
<td>1061</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>2511</td>
<td>1589</td>
<td>1078</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>2601</td>
<td>1629</td>
<td>1095</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>2692</td>
<td>1669</td>
<td>1112</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>2785</td>
<td>1709</td>
<td>1129</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>2879</td>
<td>1749</td>
<td>1146</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>2974</td>
<td>1789</td>
<td>1163</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>3070</td>
<td>1829</td>
<td>1180</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>3168</td>
<td>1869</td>
<td>1197</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>3266</td>
<td>1909</td>
<td>1214</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>3366</td>
<td>1949</td>
<td>1231</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>3468</td>
<td>1989</td>
<td>1248</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>3570</td>
<td>2029</td>
<td>1265</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>3674</td>
<td>2069</td>
<td>1282</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>3779</td>
<td>2109</td>
<td>1298</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>3885</td>
<td>2149</td>
<td>1314</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>3992</td>
<td>2189</td>
<td>1330</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>4101</td>
<td>2229</td>
<td>1346</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>4210</td>
<td>2269</td>
<td>1362</td>
<td></td>
</tr>
</tbody>
</table>

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


