An Initial Proposal for Data-Aware Resource Analysis of Orchestrations with Applications to Proactive Monitoring

December 2009

facultad de informática

universidad politécnica de madrid

Dragan Ivanović
Manuel Carro
Manuel Hermenegildo

TR Number CLIP 6/2009.0

Technical Report Number: CLIP 6/2009.0
November, 2009

Authors

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

Keywords

Service Orchestrations, Resource Analysis, Data-Awareness, Monitoring

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483. Manuel Carro and Manuel Hermenegildo were also partially supported by Spanish MEC project 2008-05624/TIN *DOVES* and project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo was also partially supported by EU projects FET IST-15905 *MOBIUS*, FET IST-231620 *HATS*, and 06042-ESPASS.

Abstract

Several activities in service oriented computing, such as monitoring, automatic composition, and adaptation, can benefit from knowing ahead of time future properties of a given service composition. In this paper we focus on how statically inferred cost functions on input data, which represent safe upper and lower bounds for different cost measures, can be used to predict some runtime QoS-related values (to, e.g., validate compositions at design time) and to compare actual and predicted resource usage at run-time in order to take adaptive actions if needed. In our approach a BPEL-like orchestration is expressed in an intermediate language which is in turn automatically translated into a logic program. Cost and resource analysis tools are applied to infer functions which, depending on the contents of some initial incoming message, return safe upper and lower bounds of some resource usage measure.

Contents

1 Introduction

Service Oriented Computing (SOC) [1] is a well-established paradigm which aims at expressing and exploiting the computation possibilities of remotely interacting loosely coupled systems that expose themselves using service interfaces whose description may include operation signatures, behavioral descriptions, security policies, and other features, while the implementation is completely hidden. Several service interfaces can be *put together* to accomplish more complex tasks through the so-called *service compositions*. Such composition is usually written using a general-purpose programming language or some language specifically designed to express SOC compositions [2, 3, 4]. Service compositions, in turn, can expose 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 makes it necessary 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 (which may involve, e.g., rebinding to different services with compatible semantics and better behavior) may become necessary. When deviations are predicted ahead of time instead of detected when they happen, the system is performing proactive monitoring. This is, of course, more complex but also more interesting and useful, as it performs *prevention* instead of *healing*.

Monitoring usually requires *snooping* the actually delivered quality of service (QoS) in order to detect (undesired) underperformance with respect to the planned execution. However, comparing actual and expected QoS of a composition —even assuming the composition does not change over time— is far from trivial. Clearly, the more accurately one can calculate the expected QoS, the better predictions can be made. In estimating QoS behavior, two factors, at least, have to be considered:

- The structure of the composition itself, i.e., what it does with incoming requests and which other services it invokes and how (which is, initially, under the control of the designer or, at least, completely known at any moment in time), and
- The variations on the environment, such as network links going down or external services not meeting the expected deadlines, which are contingent and usually out of control.

As an example, when predicting the total time spent in sending and receiving messages one must take into account, on one hand, the number of service invocations in each direction (which depends on the structure of the composition) and, on the other hand, the time sending and receiving every message takes, which is outside the control of the composition.

Of these two sources of information, the latter has been extensively studied [5, 6, 7, 8], while the former has been, to our knowledge, less deeply explored. In particular, certain characteristics of some SOC-oriented languages, like fault handling, different patterns for message-based invocation, etc. have not been appropriately taken into account: often, problematic constructs of the language under study were ignored. Also, information such as the actual data received through a service invocation has been recognized as relevant [9, 10] but has not been correctly addressed so far. As we will see in Section 2, the actual message contents can greatly influence the runtime behavior of a composition (e.g., reserving hotels for one person is, from the point of view of spent resources, not the same as reserving for one hundred, since more messages are sent, more bandwidth is spent, etc.), which makes prediction techniques that do not take run-time parameters into account potentially inaccurate.

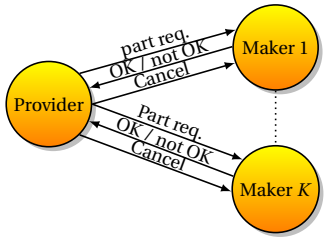

Figure 1: Simplified hotel reservation system.

In this paper we will focus on developing a methodology, based on previous experience on automatic complexity analysis [11, 12, 13], which can generate correct approximations of cost functions measuring a variety of relevant execution characteristics via translation to an intermediate language (Sections 3, 4, and 5). These functions use (abstractions of) incoming messages in order to derive correct upper and lower bounds which depend on the input data and which are potentially more accurate that data-unaware approximations. In Section 6 we show how these functions can be used to help monitoring make better decisions.

We want to note that correct data-aware cost functions can in general be applied to any situation where a more informed QoS estimation is an advantage. In particular, QoS-driven service composition [14, 15, 16] can use them in order to select better service providers given information on which kind of requests are expected. In a related setting, adaptation mechanisms can also benefit from such a knowledge [17].[1]

2 A Motivating Example

We illustrate with a simple example how actual data can be taken into account when generating QoS expressions for service compositions.

Example 1 *Figure 1 shows a simple hotel reservation system. The Client (e.g., a browser maybe operated by a final user or by a travel agency) gets in touch with a Booking Agency and requests N hotel rooms. The Booking Agency runs (or accesses) a composite service which tries a number K of hotels until it either finds all N rooms, or replies with a* no rooms available *message. Moreover, the service books rooms one person at a time as they are available, and, if after scanning all the hotels, not enough rooms are available, it revokes the reservations made so far by means of cancellation messages. A hotel that reports that it has no more available rooms is excluded from further search. We assume that one message is used to for each room query, one for each confirm / reject reply, and one for each reservation revocation.*

Note that it is unlikely that the whole process can be made as a single transaction because the reservation system of the different hotels may very well be disconnected; therefore it has to be instrumented at the level of composition.

We will assume that we are interested in the number of messages sent / received. There are several reasons for this: in a real system, message exchange can carry a sizable overhead, thus significantly affecting the actual execution time; it is possible that hotel reservation services take a toll on every

[1] Our related work on applying the derived cost functions to guide adaptation [17] involves (re)binding of service candidates on that basis. Although the technique for deriving cost bound functions is the same, we here focus on the different problem of application to proactive monitoring.

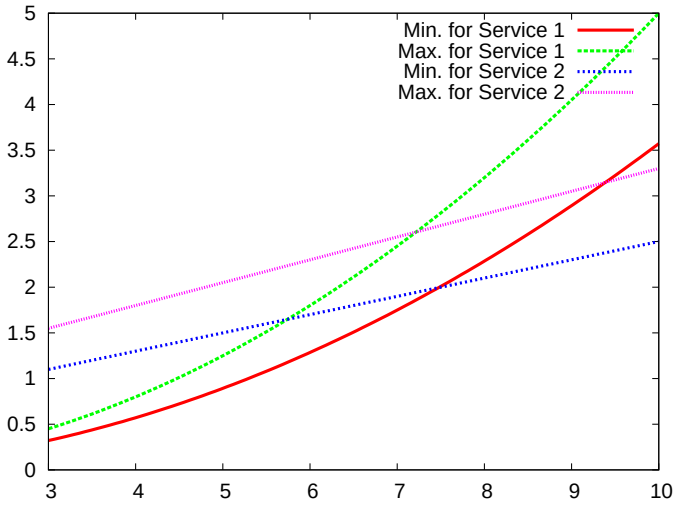

Figure 2: Upper and lower bounds for two services.

message they answer; and the Booking Service could also charge some amount of money per message.

Assuming $K \geq N$, the *minimum* number of messages that can be sent before returning is $2N$, corresponding to a successful reservation (N successful requests and replies to the same hotel) while the *maximum* number of messages is $2K + 3(N-1)$, corresponding to the worst case unsuccessful reservation ($N-1$ successful reservations, plus one last unsuccessful reservation which triggers cancellation of the $N-1$ successful reservations). Between these extremes, the maximum for a successful reservation would be $2K + 2(N-1)$ messages.

The analysis is not trivial, even for this very simple case, and depends, on one hand, on the internal logic of the composition and on the other hand on the values of N and K, which should be considered parameters for the composition, since it is more likely that the hotels are listed in a separate registry, than hardwired into the composition code.

Compared with probabilistic approximations, the following differences can be pointed out:

- In the dataless formulation, the impact of loops and conditionals can at most be estimated based on, for example, historical data. It cannot be used to actually give any *guarantee*, as the value for any QoS characteristic will be constant regardless of the actual input values for K and N.
- Additionally, safe upper and lower approximations (e.g., bounds) cannot be usually obtained, as these probabilistic formulae only use a single number representing some type of average.
- In the case of QoS-aware matchmaking or rebinding, comparing two different service compositions ignores the functional dependency that the QoS has on the data. Figure 2 portrays the

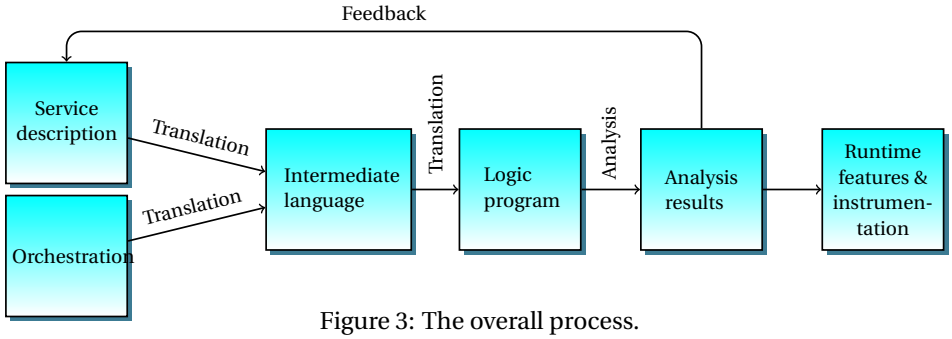

Figure 3: The overall process.

upper and lower bounds of two compositions for some QoS as a function of a single input parameter. For same ranges of data input one composition is preferable over the other, while in the central, *shared* zone the information we have is not enough to decide. Knowing in which area of the graph we are located is relevant in order to make the best matching choice.

We primarily aim at inferring functions counting a number of relevant *events*.[2] To this end, we follow the approach to resource-oriented analysis of [18, 19]. The fundamental idea is to specify how much every part of a composition contributes to the usage of some resource, and derive cost functions based on that specification. A key aspect is that, for operations which we cannot analyze (because they are external to the composition process, such as database accesses) but whose cost we want to take into account, the specifications of how much resources they consume can be made through functions that take into account the actual data.

3 An Overall Description of the Analysis Process

Figure 3 shows the overall picture of the process we present. The orchestration description (which can be written e.g. in BPEL, although our approach should be valid for other orchestration languages), together with meta-information usually contained in the related WSDL document, is translated into an intermediate language whose constructs are shown in Table 1.

The declarations in Table 1 describe namespace prefixes (used for qualified names), XML-schema-derived data types for messages, and service port types. Besides, the intermediate language allows declaring external services that are not analyzed, but have some trusted properties that are either results of a separate analysis or *a priori* assumptions.

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. The choice of activities in Table 1 is driven by the key features of the subset of BPEL we are concerned with. In particular, we restrict ourselves to orchestrations that accept a single input message and terminate their work by either dispatching a reply or failing. That is by far the most common type of orchestration, although extension to orchestrations that may accept several different input messages is straightforward using the native non-determinism available in the target platform (see below). Support for resource analysis of stateful service callbacks is a subject for future work.

Next, the intermediate representation is translated into a logic programming language (Ciao [20])

[2] Note that the technique we are building on was primarily applied to compute execution steps, which are close to execution time.

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.

augmented with assertions [21] which allow expressing types and modes (i.e., which arguments are input or output) as well as resource definitions and functions describing resource consumption bounds. The type and mode assertions help the analyzer to "understand" more precisely what the original program meant —i.e., not to lose the information about data directionality that was present in the original orchestration. Intuitively, the reason to do this is that since Prolog has a very free view of types and a complex control strategy (including built-in backtracking), a naïve, unannotated translation would generate a program exhibiting more possible behaviors than those of the original BPEL program, and therefore the analysis results would very likely lose precision. The logic program resulting from the translation is fed to the resource consumption analyzer of the Ciao preprocessor (CiaoPP), which is able to infer upper and lower bounds for the generalized cost / complexity of a logic program [11, 12, 18, 19].

The results of the analysis (the cost functions) are fed back to service description, thus adding more information on the service, which can be stored in a registry and used by cost-sensitive binding and matchmaking algorithms. The results are also used to inform the infrastructure and monitoring parts of the SOC architecture on the expected runtime features of the orchestration and thus help deployment, compilation into object code, and run-time instrumentation.

An important observation regarding the translation is that, in general, we do not need the generated logic program to be strictly faithful to the operational semantics of BPEL: it has to reflect just the necessary part of the semantics that will 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 closely the operational semantics of the BPEL process under analysis.

4 An Outline of the Translation from BPEL to Logic Programs

In this section we will briefly describe the translation of BPEL process definitions, via the intermediate language, to a logic program that is analyzed by existing tools. A set of BPEL processes which form a (small) service network are taken as the input to the process and the result is a single file with a logic program, where BPEL processes are mapped onto predicates which call each other when the original BPEL processes would invoke another service. In order for the final code to be amenable to analysis, we currently restrict ourselves to a subset of BPEL, which notwithstanding we consider rich enough to express an ample class of interesting real-life cases.

4.1 Restrictions on Input Orchestrations and Correspondence with BPEL

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. 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 in the same way we now treat whole orchestrations.

The activity constructs in the intermediate language in Table 1 are inspired by the key features of BPEL but are applicable to other abstract or executable orchestration languages. Some activity constructs (`empty`, assignment, sequence,...) are commonly found in programming languages. The key constructs for modeling orchestration workflows are `flow`, `float`, `scope/handler`, and `invoke`.

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.

The main purpose of `scope` constructs in BPEL is to introduce local variables, fault, and compensation handlers. In our intermediate language, `scope` serves this purpose, with the exception of compensation handlers, which we do not directly support. Compensation handlers in BPEL contain logic that "undoes" effects of a successfully completed scope. As such, a compensation handler is a pseudo-subroutine attached to a scope, which must be explicitly invoked from a fault handler or compensation handler of an enclosing scope. However, the BPEL specification requires compensation handlers to operate on a snapshot of the scope's variables made on successful completion of a scope, including each individual iteration of loop body, which introduces considerable problems for the analysis as it is now. If we ignore value snapshots, BPEL compensation handlers can be inlined at the place of their invocation.

4.2 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 the

```
:- regtype 'acme->reservationData'/1.
'acme->reservationData'('acme->reservationData'(A, B, C)):-
    num(A), num(B), list(C, 'acme->personInfo').

:- regtype 'acme->personInfo'/1.
'acme->personInfo'('acme->personInfo'(A, B)):-
   atm(A), atm(B).
```

Figure 4: Translation of types.

purpose of simplicity, we abstract the multitude of simple types in XML Schemata into just three disjoint types: `numbers`, `atoms`, representing strings, 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 4 shows an automatically obtained actual translation for the hotel reservation scenario in Example 1. The type name `'acme->reservationData'` is a structure with the same name and with three fields: two numbers and a list of elements of type `'acme->personInfo'`. Each of these elements is in turn a structure with two fields being an atom each.

Following BPEL, 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`. A set of standard XPath operators and basic functions is supported, including `position()` and `last()`.

To assist the analyzer in tracking component values and correlating the changes made to them, we take the approach of statically decomposing lists and XML structures in an execution environment into their components, and passing them around explicitly as predicate arguments from that point onwards. Unfolded structures no longer need to be passed along with components, since they can be reconstructed on demand. The resulting code is less readable for a human, but more amenable to analysis.[3]

For instance, to access the third element of a list stored as an opaque object in a variable, the list has to be decomposed into head and tail subcomponents, and the process has to be repeated until the third element of the list is reached. From that point, the list can be reconstructed on demand from the first three elements and the remainder, and therefore need not be explicitly passed in predicate calls. However, if the list is assigned (from an expression or by receiving a reply message), we cannot guarantee any more that it has at least three elements, and therefore the list once again becomes an opaque object. The same logic applies to other data structures and their components.

4.3 Basic Service and Activity Translation

The basic idea of the automatic translation from the intermediate language is to keep track of the functional dependency of the resulting response message on the input message with which a service is invoked. Here we present the translation scheme based on generation of clauses of a logic program [17] that can be automatically analyzed for resource usage. An orchestration S is translated

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

into a predicate:

$$s(\bar{x}, y) \leftarrow [\![A]\!]_{\eta_0}(y)$$

where $\bar{x}$ represents components of the input message, y stands for the response message, and $[\![A]\!]_{\eta_0}(y)$ is the translation of the orchestration body A with respect to the initial service environment η_0. An environment maps symbolic (sub)component names (which denote message parts, nested XML elements and attributes, and scalars) to logical terms. Each variable in the environment is either a scalar or a tree-like structure where component nodes branch from structure nodes, up to some depth of unfolding, as explained in the previous subsection. Unfolded structures in an environment (the internal nodes) can always be recursively reconstructed from their components (children nodes). Consequently, the entire environment can be represented by the leaf nodes. When η appears in an argument position, it stands for the list of leaf nodes in η. Leaf nodes of the initial environment η_0 are the list $\bar{x}$ of input message components.

The translation operates on a non-empty sequence of activities, which we can write as $\langle A|C\rangle$, where A is the first activity, and C is the continuation sequence, which may be empty (ε). We write $[\![A|C]\!]$ to denote the translation of $\langle A|C\rangle$, and, as a shorthand, $[\![A]\!]$ to denote translation of $\langle A|\varepsilon\rangle$. This allows us to normalize translation of a sequence (A_i, A_j) 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 the reply contained in variable v in the current environment:

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

Raising a fault with `throw` is translated into a logical failure ($[\![\texttt{throw}|C]\!]_\eta(y) \equiv \texttt{fail}$), which can be caught on backtracking by fault handlers. The `empty` activity is ignored, so that $[\![\texttt{empty}|C]\!]_\eta(y) \mapsto [\![C]\!]_\eta(y)$.

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

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

where clauses generated for a_i depend on A_i, η, and C. First we look at the case when $A_i \equiv x{<}{-}e$, i.e., the XPath expression e is evaluated and assigned to the environment element x (a variable or its component). The generated clause has several segments:

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

where $[e : E]_\eta$ stands for evaluation of e into term E in the environment η, and $[E/x]_\eta^{\eta'}$ stands for mutation of η into η' as the result of assigning E to x. Likewise, in case of an external service invocation, $A_i \equiv \texttt{invoke}(p, o, v, w)$, the generated clause has the form:

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

where s_{po} is the translation of a service implementing operation o on port type p, variable v holds the input message, and variable w receives the reply. For $A_i \equiv \texttt{if}(c, A_j, A_k)$, two clauses are generated:

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

where $[c?]_\eta$ stands for code that succeeds if and only if the boolean condition c evaluates to `true`. On the basis of `if`, we generate recursive clauses for the case $A_i \equiv \texttt{while}(c, A_j)$:

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

Note how reappearance of A_i in the first clause leads to a recursive definition of the translation scheme. The above translation is however not circular, because we already know that $[\![A_i | C]\!]_\eta(y) \equiv a_i(\eta, y)$. Other looping constructs, such as `repeatUntil` and `forEach` reduce to `while`.

4.4 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 [22], 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. A $\texttt{float}(D, A)$ construct appearing in the body of a `flow` uses attributes D to annotate activity A with input link dependencies and output transition. Links are internally declared as Boolean variables. The floating activities are ordered so that the link dependencies are respected. As in BPEL itself, there can be no circular link dependencies. After reordering, a `flow` effectively translates to a sequence, and each $\texttt{float}(D_j, A_j)$ is transformed into:

$$\texttt{if}(c_j, (A_j, \texttt{'\$}o\texttt{' <- 'true()'}), \Phi)$$

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

4.5 Accounting for Unavailable Code

So far we have assumed that the analysis operates on a static composition whose code is available. The same approach can be easily extended to the case where we have a collection of interacting compositions, with statically available code, whether or not these compositions are expected to be deployed locally or remotely. However, there are cases where such code may not be available such as, for example, when some provider does not want to reveal which code is being run on its servers. In such scenarios it is still possible to exploit the partial statically inferred resource usage information to drive cost-sensitive adaptation [17].

Note also that code disclosure concerns may not present a problem for static analysis. The analysis, while starting with an executable (e.g. BPEL) code, does not actually act on such code directly,

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

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

:- struct( hotres:resData, [
       child( hotres:personCount): number,
       child( hotres:priceLimit): number,
       child( hotres:person):
 list( struct( hotres:persInfo)) ]).

:- struct( hotres:persInfo, [
       attribute( '':firstName): atom,
       attribute( '':lastName): atom,
       child( hotres:hotelName): atom,
       child( hotres:roomNo): number ]).

:- port( hotres:agency, [
       reserveGroup( struct( hotres:resRequest)):
  struct( hotres:resResponse) ]).

:- port( hotres:hotel, [
       reserveSingle( struct( hotres:persInfo)):
                      struct( hotres:persInfo),
       cancelReservation( struct( hotres:persInfo)):
 struct( hotres:persInfo) ]).
```

```
service( hotres:agency, reserveGroup, '$req', '$resp'):-
[
 '$resp.body/hotres:personCount'<-0,
 '$resp.body/hotres:person'<-'$req.body/hotres:person',
 scope( [i:number],
 [   '$i' <- 1,
   while( '$req.body/hotres:personCount>0',
   [
     scope( [p: struct( hotres:persInfo),
     r: struct( hotres:persInfo)],
     [   '$p'<- '$req.body/hotres:person[$i]',
       invoke( hotres:hotel, reserveSingle, '$p', '$r'),
       if( '$r/hotres:roomNo>0',
           '$resp.body/hotres:person[$i]'<-'$r',
           throw( hotres:unableToReserveGroup) ),
       handler(
       [   while( '$i>1',
         [   '$i'<- '$i - 1',
             '$p'<- '$resp.body/hotres:person[$i]',
             invoke( hotres:hotel, cancelReservation,
                     '$p','$r')]),
             throw( hotres:unableToCompleteRequest) ])
     ]),
     '$i' <- '$i+1',
     '$req.body/hotres:personCount' <-
         '$req.body/hotres:personCount - 1' ]) ]) ].
```

Figure 5: Abstract representation of a group booking process

but rather on some abstraction in the intermediate language which can hide some details. Providers may offer this abstract code in order for third parties to check the complexity claims of the providers. By doing so they would increase the confidence of their clients without revealing more than strictly necessary. In other cases, while even this code may not be available, the owner of the service can provide sufficient information in the form of resource assertions which describe the resource consumption behavior without disclosing code in the least.

5 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 hotel booking, along the lines (but slightly simplified, for space reasons) of the example used in Section 2. For compactness, we present the abstract description of this orchestration in our internal representation form instead of plain BPEL (Figure 5). 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 people to book a room for and tries to reserve a room in a hotel by invoking an external hotel service.[4] If that is not possible, or if a failure arises, a failure

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

```
:- entry 'service_hotres->agency->reserveGroup'/4
 :{gnd,num}*{gnd,num}*{gnd,'list_of_hotres->persInfo'}*var.

'service_hotres->agency->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/hotres:personCount>0')),
 A>0, !, act_5( A, B, C, D, E, F, G, H).
act_4( _, _, _, D, E, F, _, 'hotres->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( hotres:hotel, reserveSingle, '$p', '$r')),
 H='hotres->persInfo'(N, O, P, Q),
 'service_hotres->hotel->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 6: Translation into parts of a logic program.

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 5. 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 ω. 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 this example are: the number of all basic activities performed (assignments, external invocations); the number of invocations of individual room reservations (operation `reserveSingle` at the hotel service); and the number of invocations of reservation cancellations (operation `cancelReservation` at the hotel service). From the number of invocations it is easy to deduce the number of messages exchanged during the execution of the process: a single reservation counts as two, and a cancellation counts as one message. The results are displayed in Table 2, where the estimated upper and lower bounds are expressed as a function of the initiating request.

We model processing of a single reservation failure with fault handling that interrupts the normal (nominal) flow and triggers cancellations. We differentiate explicitly between the case with costs of fault processing included, which gives wider, more cautious estimates, and the case in which the execution is successful (i.e., without fault generation and handling). These two cases were obtained

	With fault handling		Without fault handling	
Resource	lower bound	upper bound	lower bound	upper bound
Basic activities	2	$7N$	$5N+2$	$5N+2$
Single reservations	0	N	N	N
Cancellations	0	$N-1$	0	0
No. of messages	0	$3N-1$	$2N$	$2N$

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

Table 2: Resource analysis results for the group reservation service

by means of different translations which explicitly generated or not Prolog code corresponding to the fault handling. The results in Table 2 correspond to best and worst case estimates from Example 1 . The upper bound on number of messages with fault handling $3N-1$ corresponds to $2K+3(N-1)$ with $K=1$, and the lower bound for the case without fault handling is $2N$.

6 Cost Functions for Monitoring

As briefly discussed in Section 1, the expected value of some QoS characteristics can be derived from the value of some cost functions and the (expected) value of some environment characteristics. In this section we will elaborate on that point and we will sketch how the availability of cost functions can be used to perform proactive monitoring.

6.1 QoS Metrics and Cost Functions

The precise cost function which is needed to express some QoS characteristic depends on the QoS metric itself. For example, if bandwidth consumption is involved in the measure of some QoS, then the number of messages and size of each message is relevant, but the number of executed activities is not directly relevant (although possibly related). However, the cost function by itself cannot in general convey all the information necessary to represent a QoS function: some data which come from the environment is needed. Therefore, and for some QoS metrics, an interval of lower and upper bounds depending on the input data can be expressed as

$$QoS_{\langle L,U\rangle}(n) = \langle cost_L(n) \oplus env_L,\ cost_U(n) \oplus env_U\rangle \tag{1}$$

where the left and right components of the tuple are the expected lower and upper bounds for the quality of service, $cost_X(n)$ is some suitable analytically determined cost / resource consumption function, env_X represents the minimum and maximum influence of the environment conditions on the QoS at hand, and $\oplus$ is an operation which combines together the cost functions and the environment conditions.

For example, in case of the execution time of a single process, $cost_X(n)$ can be the number of activities executed, and env_L and env_U the maximum and minimum time a single activity can take (which depends on the machine executing it, the executing engine, the operating system, etc.), and $\oplus$ would be just multiplication. Since $cost_L(n)$ and $cost_U(n)$ are, respectively, the lower and the upper bound, then if we assume that env_L and env_U are also correct lower and upper bounds, the calculated QoS will be a correct lower and upper bounds of the actual (runtime) QoS values.

Note that this generic scheme can admit variations: for example a more accurate approximation

of execution times can be inferred by assigning a different weight to each type of activity. In this case, env_X would actually be an array with a component for the execution time for every type of activity, $cost_X(n)$ would also be an array counting how many times every type of activity is executed, and $\oplus$ would be the vector dot product.

6.2 QoS and Cost Functions During Composition Execution

Given some QoS characteristic which we assume fixed, Equation (1) relates it to a cost function. It is always the case that the general cost function of a composition is made up of several parts, each one referring to a structural part of the composition. As an example, the upper bound of the cost of an `if-then-else` construct in terms of, for example, executed activities, is the upper bound of the condition plus the maximum of upper bounds of the `then` and the `else` parts.

We can associate to every program point a measure of how much cost / resources remain to be spent in the rest of the execution.[5] For example, in the `if-then-else` example before, once the `if` part is done, what remains is either the `then` or the `else` part. In general this measure depends not only the point in the service composition, but also on the values of the data at that moment: for example, in a loop, where the same activity is executed several times, less "cost" is left until the end of the execution after every iteration, even in the same point of the composition. The difference comes from the different state of the variables, and this is one of the reasons why taking data into account is beneficial.

There is, therefore, a notion of "pending" QoS, which comes from using these composition-point cost functions together with the environment characteristics: for example, from the activities remaining to be executed and the expected time of every activity, the remaining time to the completion of the service activation would be a "pending" QoS. This is, of course, interesting from the monitoring point of view. Assuming that a composition has been designed and approved on the basis of the expected lower and upper bounds of QoS (i.e., the required QoS adequately falls within these bounds), then deviations of the environmental characteristics can be used to predict more accurately what will be the QoS at some future point by dynamically combining the cost / resource consumption functions (which we are assuming do not change during service execution) with the actual environment conditions, which may deviate from those initially assumed.

Figure 7 exemplifies such a situation. Let us assume we are interested on some QoS metric of a composition, whose value must not go over some limit *Max*. Therefore, we should use cost functions and environment characteristics representing safe upper bounds: if the upper bound is smaller than some limit, then we have the guarantee we need.[6] Symmetrically, if we are concerned with a QoS attribute whose value should not go below some minimum, we would use lower bounds instead. We designate four points ($\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$, and $\mathscr{D}$) in the execution of some composition and we will focus on how monitoring at these points can be done proactively with the use of cost functions. In Figure 7 the solid line represents the initial running QoS predicted taking into account the statically inferred cost functions and the expected environment conditions. The dashed line represents the actual observed QoS.

At point $\mathscr{B}$, the actual quality has deviated with respect to the predicted one. Since the composition has not changed, and thus neither have the cost functions, we can conclude that the deviation

[5] In fact the initial cost is just a measure of how much it remains to be spend assuming that, since the execution has not started, a total of 0 has been spent.

[6] For completeness: if the upper bound ends up over the limit, but the lower bound does not, there is a possibility that things go wrong. If the lower bound goes over the limit, we have the guarantee that the execution is going to violate the initial restrictions. We will assume we do not want to run any risk.

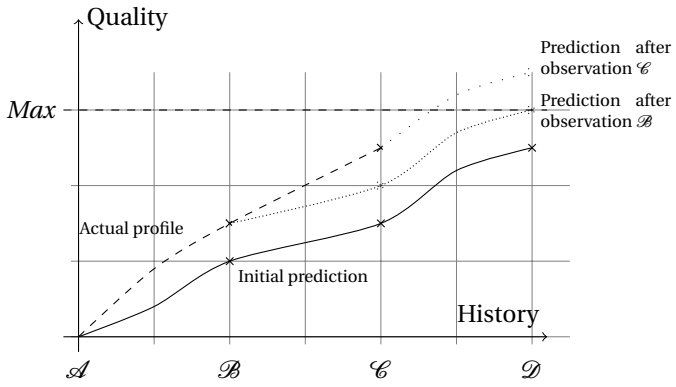

Figure 7: Actual and predicted QoS throughout history.

can only be due to a change in the environment behavior (e.g., additional load on a server or a faulty network). A new prediction for the future can be done by using the observed influence of the environment so far and the existing cost function. This new prediction curve (densely dotted) still ends, at point $\mathscr{D}$, within the limits of the acceptable range *Max*. However, at point $\mathscr{C}$ a new observation gives yet higher values for the QoS value and, therefore, for the influence of the environment. Yet another function and associated plot curve (sparsely dotted) can be constructed which predicts that it there is the possibility that the execution finishes violating the QoS constraints. Therefore, at point $\mathscr{C}$ we can raise an alarm and maybe trigger an adaptation procedure. Note that we in fact have detected a problem before it actually appeared. In order for this technique to work in complex service compositions with loops, different response times depending on invocations, etc. it is necessary to take data into account from the beginning.

7 Conclusions and Future Work

We have presented a resource analysis for service orchestrations (which we instantiate to the BPEL case) which is based on a translation to an intermediate programming language (Prolog) for which complexity analyzers are available. These analyzers can be customized to deal with user-defined resources, thereby opening the possibility of generating resource-consumption functions, some of them of interest for SOC. Inferring these functions can be used as core technology for some approaches to proactive monitoring, 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 (the cost functions) are valid for the original network. We have sketched a mechanism to use these functions, together with environmental characteristics, to predict the future behavior of the system even when the environment deviates from its expected behavior.

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 of this (and other) issues is the subject of current work.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Service-Oriented Computing. Communications of the ACM **46**(10), pp. 24–28 (2003)

2. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: 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)

3. Zaha, J.M., Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Let's Dance: A Language for Service Behavior Modeling. In: OTM Conferences (1). pp. 145–162. (2006)

4. van der Aalst, W., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In Leymann, F., Reisig, W., Thatte, S.R., van der Aalst, W., eds.: The Role of Business Processes in Service Oriented Architectures. Number 06291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2006)

5. Mukherjee, D., Jalote, P., Nanda, M.G.: Determining QoS of WS-BPEL Compositions. In: ICSOC. pp. 378–393. (2008)

6. Wu, J., Yang, F.: A Model-Driven Approach for QoS Prediction of BPEL Processes. In: ICSOC Workshops. pp. 131–140. (2006)

7. Buccafurri, F., Meo, P.D., Fugini, M.G., Furnari, R., Goy, A., Lax, G., Lops, P., Modafferi, S., Pernici, B., Redavid, D., Semeraro, G., Ursino, D.: Analysis of QoS in Cooperative Services for Real Time Applications. Data Knowledge Engineering **67**(3), pp. 463–484 (2008)

8. Fugini, M.G., Pernici, B., Ramoni, F.: Quality Analysis of Composed Services through Fault Injection. In: Business Process Management Workshops. pp. 245–256. (2007)

9. Cardoso, J.: 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. pp. 67–74. (2005)

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

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

12. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.W.: Lower Bound Cost Estimation for Logic Programs. In: 1997 International Logic Programming Symposium, pp. 291–305. MIT Press, Cambridge, MA (October 1997)

13. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: 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. Canfora, G., Penta, M.D., Esposito, R., Villani, M.: An Approach for QoS-Aware Service Composition Based on Genetic Algorithms. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, New York, NY, USA, pp. 1069–1075. ACM (2005)

15. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. Software Engineering, IEEE Transactions on **30**(5), pp. 311–327 (May 2004)

16. ping Chen, Y., zhi Li, Z., xue Jin, Q., Wang, C.: Study on QoS Driven Web Services Composition. In: Frontiers of WWW Research and Development - APWeb 2006. Volume 3841 of Lecture Notes on Computer Science., pp. 702–707. Springer Verlag (2006)

17. Ivanović, D., Carro, M., Hermenegildo, M., López, P., Mera, E.: Towards Data-Aware Cost-Driven Adaptation for Service Orchestrations. Technical Report CLIP5/2009.0, Technical University of Madrid (UPM) (November 2009)

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

19. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: 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, pp. 154–168. Springer-Verlag (August 2007)

20. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Morales, J., Puebla, G.: An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In Pierpaolo Degano, Rocco De Nicola, J.M., ed.: Festschrift for Ugo Montanari. Number 5065 in LNCS. Springer-Verlag (June 2008) pp. 209–237

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

22. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: 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)