Towards Data-Aware Resource Analysis for Service Orchestrations*

Dragan Ivanović[1], Manuel Carro[1], and Manuel Hermenegildo[1,2]

[1] School of Computer Science, T. University of Madrid (UPM)
(idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es)
[2] IMDEA Software, Spain

Abstract. Compile-time program analysis techniques can be applied to Web service orchestrations to prove or check various properties. In particular, service orchestrations can be subjected to resource analysis, in which safe approximations of upper and lower resource usage bounds are deduced. A uniform analysis can be simultaneously performed for different generalized resources that can be directly correlated with cost- and performance-related quality attributes, such as invocations of partners, network traffic, number of activities, iterations, and data accesses. The resulting safe upper and lower bounds do not depend on probabilistic assumptions, and are expressed as functions of size or length of data components from an initiating message, using a fine-grained structured data model that corresponds to the XML-style of information structuring. The analysis is performed by transforming a BPEL-like representation of an orchestration into an equivalent program in another programming language for which the appropriate analysis tools already exist.

Keywords: Service Orchestrations, Resource Usage, Program Analysis, Data-Awareness

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 remotely interact. In any case, they 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 service interfaces can be *put together* to accomplish tasks more complex than any of them in isolation through the so-called *service composition*. This composition is usually written in a form of programming language which can be designed ad-hoc

* 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.

for SOC compositions [1, 2] or general-purpose languages. These compositions, in turn, expose themselves as full-fledged services.

Among the key distinguishing features of SOC w.r.t. other classical, conventional programming paradigms, SOC systems are expected to live and be active during long periods of time and span different geographical and administrative (also in the sense of "computer administration") domains. Moreover, they may need to provide transactional behavior, late binding, undergo evolution without stopping the system, be able to monitor and adapt themselves to changing environment conditions (including decreasing quality of service (QoS) or even malfunctioning in parts of the system), (re)assemble automatically different services to meet some functionality requirement, and other characteristics which make SOC an incredibly challenging area full of open research and development challenges.

One of these challenges is the selection of services given a set of requirements, which can address what the service is actually required to do, and in which terms of speed, throughput, reliability, etc. A service description, in turn, asserts what the service offers in terms of interface, behavior, QoS, etc. Pairing the demands of a service composer with the offers of several services is known as *matchmaking*. While functional matchmaking is of course necessary, as the number of available services grows it is expected that equivalent (in the functional sense) services will have to be decided upon by looking at, e.g., their QoS or other characteristics. Therefore, QoS is now regarded as an integral part of the composition of services [3–5], and is regarded as a contract which has to be enforced. Such a contract is commonly implemented as a Service Level Agreement (SLAs) [6] which specifies the relevant QoS dimensions, including the level of QoS attributes, and may also include details of QoS assurance, party responsibilities, and actions to be taken if the actual QoS does not meet the SLA conditions.

Several QoS characteristics have been under study [7], many of them related to events that can only be probabilistically predicted or, at most, based on past history plus, for example, data mining and prediction techniques, such as response time, availability, reliability, etc.[3] Others have a well-defined value at every moment in time, but which can change nonetheless along history, such as cost of a service per invocation. Finding techniques to automatically generate the expected QoS for a service composition, which presents itself as a service, is a non-trivial, very important task in order to automate the creation and self-management of SOC systems.

Several challenges stem from this. One is to actually traverse the structure of every single service composition, or service composition candidate, so that the QoS characteristics from each of the invoked services is taken into account. While somewhat abstract formulation based on workflow patterns appear in the relevant literature [8, 9] (see Table 1, taken from [8] as an easy-to-understand example), they fall short in at least two points:

– Actual languages for composition, such as BPEL and, of course, any general-purpose language, have a richer set of control structures, which include stopping parallel tasks, implementing compensation handlers, and managing exceptions.

[3] Note that we are, in principle, not dealing with Qos characteristics such as security policy which are more difficult to quantify.

QoS Attr.	Sequence	Loop	Switch	Flow
Time (T)	$\sum_{i=1}^{n} T(A_i)$	$k * T(A_1)$	$\sum_{i=1}^{n} p_i * T(A_i)$	$\max_{i=1}^{n} T(A_i)$
Cost (C)	$\sum_{i=1}^{n} C(A_i)$	$k * C(A_1)$	$\sum_{i=1}^{n} p_i * C(A_i)$	$\sum_{i=1}^{n} C(A_i)$
Availability (V)	$\prod_{i=1}^{n} V(A_i)$	$V(A_1)^k$	$\sum_{i=1}^{n} p_i * V(A_i)$	$\prod_{i=1}^{n} V(A_i)$

Table 1. Derived estimates for QoS attributes of a composite service. The body of a loop contains only one sub-activity denoted A_1, and k is the estimated number of iterations. p_i is an estimated probability of switching to branch i.

– The impact of the actual runtime data is often neglected, and is abstracted or approximated by using a probabilistic measure of the *effects* of such a data, such as number of iterations in a loop, probability of taking some branch of an if-then-else, etc.

Given that the actual message contents can greatly influence the runtime behavior of a system 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.), in this paper we will present an approach to automatically deduce QoS expressions (or, more precisely, functions which are upper and lower bounds of the possible QoS values) which take into account the runtime data which is processed by some composition.

The rest of the paper proceeds as follows: Section 2 will give some more overall details on the approach we propose, Section 3 provides some details on the analyzer and the analysis on which we technologically base our proposal, Section 4 sketches how we translate BPEL processes to the input language of the analyzer, Section 5 provides some examples of actual orchestration code and the result of the analysis, and Section 6 closes the paper with some concluding remarks.

2 Motivation

Taking actual data into account when generating QoS expressions for service compositions opens up a series of possibilities which are out of reach for the case of probabilistically determined QoS. We will illustrate this claim with a simple example.

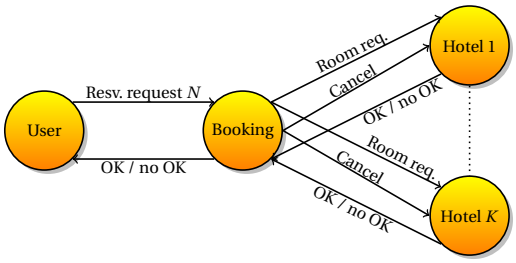

Fig. 1. Simplified hotel reservation system.

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 requesting N hotel rooms. The Booking Agency runs (or accesses) a composed service which tries a number K of hotels until it finds N rooms for all N or it replies with a* no rooms available *answer. 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. Note that 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. Besides the initial message to initiate the process, we assume a message to query for a room, a message to confirm / reject the room reservation, and another message to revoke a reservation.*

We will assume that we are interested in the number of messages sent / received. One reason for that is that in a real system connected over the Internet, message sending takes a sizable amount of the total time, thereby impacting actual time. Another reason would be that it is possible that hotel reservation services take a toll on every message they answer and, presumably, the Booking Services could also charge some amount of money per message – maybe with discounts for a large number of messages.

Assuming $K \geq N$, the *minimum* number of messages that can be sent before returning is $2+2N$, corresponding to a successful reservation (two messages to/from the Booking Service plus N successful requests to the same hotel) while the *maximum* number of messages is $1+2K+N$, corresponding to a unsuccessful reservation (initial messages plus $K-N$ unsuccessful attempts plus $N-1$ reservations plus one last unsuccessful reservation which makes the successful reservations to be undone).

Between these extremes, the maximum for a successful reservation would need $2+2K$ messages(initial messages plus $K-N$ unsuccessful attempts to reserve plus N successful reservations) and the minimum for an unsuccessful reservation is $2+2K$.

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 can be taken as parameters for the composition, as it is more realistic to think that the hotels are stored in a separate database rather than hardwired in the composition code.

Compared with the probabilistic approximation which can be generated applying formulas similar to those in Table 1, the following differences can be pointed out:

- In the dataless formulation, the contribution of loop iterations and conditionals can at most be estimated based on, for example, historical data. Therefore it cannot be used to actually give any *guarantee*, as the value for any QoS characteristic will be the same regardless of the actual input values for K and N.
- Additionally, safe upper and lower approximations (e.g., bounds) cannot be usually drawn, as these formulae only return a single number.
- In the case of QoS-aware matchmaking, comparing two different service compositions ignores the shape data which may depends on the data. The advantage of having functions available, and using it, is that a more informed decision can

be made. For example, in Figure 2, the upper and lower bounds for some QoS measure as function of a single input data are plotted. For same ranges of data input, one composition is preferable to the other, while the central *shared* zone is, in principle, almost equivalent.

Complexity analysis (either automatic or not) has often geared towards finding out functions which express (bounds of) some notion of time complexity, usually measured as computation steps needed to accomplish a task, where the basic language constructs are assumed to have a constant cost. Differences between languages are overcome by using the $O(\cdot)$ notation.

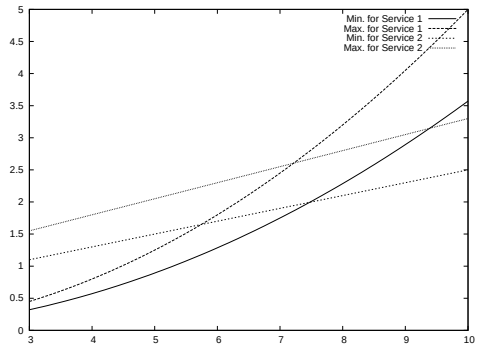

Fig. 2. Upper and lower bounds for two services.

However, time complexity (and even actual time) are, in our opinion, not really meaningful for the SOC landscape. Time complexity is too fine grain a measure, and its actual value is easily blurred by the work different software layers which take part in a SOC architecture. Likewise, actual execution time depends on so many factors (network latency, network occupation, different execution platforms, CPU load of the host where some of the application application may be being) that a theoretically precise answer is very unlikely to be applicable larger, realistic examples with a reasonable degree of accuracy.

Therefore, and although the approach we propose can be used to effectively derive time complexity, we primarily aim at applying it to other resource usage measures using the resource-oriented analysis described in [10] and following the approach in [11]. In a nutshell, the basis of this user-defined resource analysis is to give the user the ability to specify how much different sections of a program contribute to the usage of some resource, and count resource usage by means of that specification. Note specifying resource usage can take into account the actual data, so no precision is necessarily lost.

Among the different resource measures which we deem interesting for SOC we can cite:

- Number of messages sent / received across the network, depending on the input data (which greatly impacts overall response time and can help to decide among several composite services).
- Bandwidth used in the network (which takes into account the number of messages plus the actual size of every message).
- Actual cost, if every message has a different cost, maybe depending on the sender / receiver of the message, so that multiplying number of messages by a per-message unit cost may not be accurate.

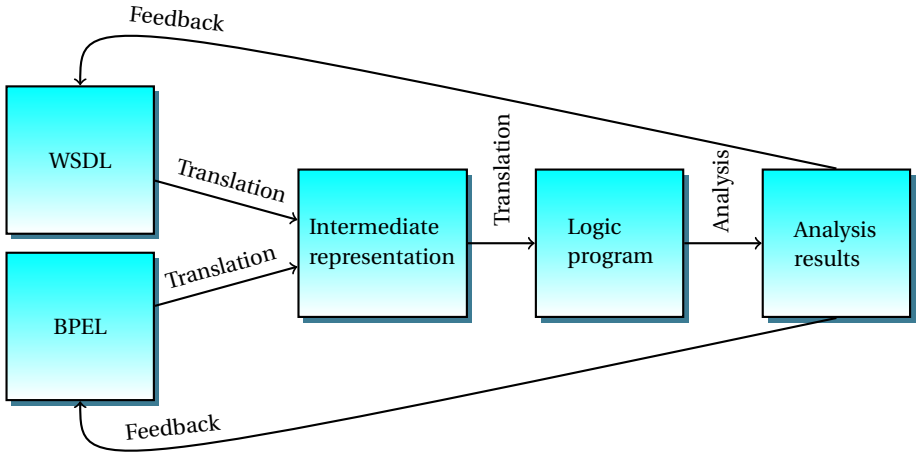

Fig. 3. The overall process.

- Number of transactions to a database; this requires marking operations which accesses the database as a resource, and tallying the number of times this resource is used.
- Number of forked processes (obviously interesting in order to limit or predict the amount of memory some network is going to take, and to detect / prevent the possibility of some ill-intentioned query will cause a real system crash).
- ...

While we will not be able to show in detail how we can account for all of these characteristics (but we will do to some extent with some of them), we hope to convince the reader that we provide the infrastructure enough, both at the conceptual and implementation level, to actually generate functions which provide safe upper and lower bounds for them, and for others not cited here.

3 An Overall Description of the Analysis Process

Figure 3 shows the overall picture of the process we have designed and implemented. The orchestration (which we are assuming is written in BPEL, although in principle the approach would be valid for other orchestration languages) is translated into an intermediate language. This language, which syntactically can be seen as Prolog terms / functors), maps very closely BPEL constructs into terms (Table 2). The advantage is that we can, later on, work directly with this representation in a very concise and comfortable way instead of having to use XML-traversal libraries which usually add a burden to the process. The same process is used to obtain an easy-to-handle intermediate representation of the XML types declared for the BPEL process. We will not go in details into this part, as it is relevant only to make the whole process easier to engineer and it adds little extra information.

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(` *PortName*`,` *Operation*`,` *InMsg*`,` *OutMsg*`)` `:-` *Activity* `.`
Activities	
Do nothing	`empty`
Assignment to variable / message part	*VarExpr* `<-` *Expr*
Service invocation	`invoke(` *PortName*`,` *Operation*`,` *OutMsg*`,` *InMsg*`)`
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*`)`
Scope fault handler	`handler(` *Activity*`)` `handler(` *FaultName*`,` *Activity*`)`
Parallel flow with dependencies	`flow(` *LinkDeclarations*`,` *Activities*`)`
Dependent activity in a flow	`float(` *Attributes*`,` *Activity*`)`

Table 2. Elements of an abstract representation of an orchestration.

This intermediate representation is, in turn, translated into a logic programming language (Ciao [12]) augmented with assertions [13] to express types and modes (i.e., which arguments are input and output). This information is generated from the original BPEL process definitions and from the XML schemata, and is meant to help the analyzer by making it "understand" more exactly what the original program meant — i.e., not to loose information existing in the original orchestration. Intuitively, the reason to do this is that Prolog has a very free view of types and a complex control strategy (including built-in backtracking) and a naïve translation would generate a program where more things than in the original one could happen, and therefore the analysis results would very likely loose precision. We currently deal with the BPEL characteristics shown in Table 2, which, for completeness, also presents the translation of the BPEL constructs into the intermediate (abstract tree) representation.

Finally, the result of the translation is to be fed into the resource consumption analyzer of CiaoPP, which is able to infer upper and lower [14, 15] bounds for the a generalization of cost complexity of logic programs.

It is interesting to note that, in general, we do not need the logic program to be analyzed to be faithful to the operational semantics of BPEL, in the sense that BPEL executions can be mimicked by the logic program. It has to reflect just the necessary semantics for the analyzers to infer information without precision loss due to the translation. This means that in general it may be necessary to tailor this translation to different cases according to the expected results of the analyzer. In our case, the translated program, while not operationally equivalent to the BPEL process, *is*

executable and does mirror the semantics of the BPEL process the domain under analysis.

4 An Outline of the Translation from BPEL to Logic Programs

In this section we will briefly describe the translation of BPEL to a logic programming language in order to analyze it taking advantage of existing tools. A set of BPEL processes which form a (small) service network are taken as input of the process and the result is a single logic programming file, where bpel process (and parts inside them) are mapped onto predicates which call each other when the actual BPEL process would invoke another service. In order to have a final code amenable to analysis, we have at the moment restricted ourselves to a subset of BPEL which notwithstanding we think is rich enough to actually express many (if not most) interesting real-life cases.

4.1 Restrictions on the Input Language

We are restricting ourselves to orchestrations that follow a `receive...reply` interaction pattern, where different activities can take place between the reception of the initiating message and the dispatching of the response. Such abstract orchestrations answer to an `invoke` from another service. We of course plan to lift this requirement in a future and accept BPEL/WSDL process descriptions, identify fragments which correspond to the request-response pattern along with the activities which *glue* them together, and treat each of these fragments as we are doing now with each BPEL process (i.e., transforming into intermediate code and analyzing them) in its entirety.

We currently do not take into account correlation sets and other run-time mechanisms for communicating between stateful instances of Web orchestrations, such as WS-Addressing. On the basis of the above mentioned fragmentation approach, part of the future work will include an extension of the translation scheme where the encapsulated state (which we have to carry around nonetheless) will be enriched to allow correct handling of service callbacks. Similarly, we do not treat compensation handlers in their BPEL semantics, as they rely on dynamic snapshots of scope variable states, but rather concentrate on modeling faults and fault handling in scopes.

4.2 Type Translation

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 themselves have simple or complex types. For the purpose of simplicity, we abstract the multitude of simple types in XML Schemata into just three disjoint types: `numbers`, `atoms`, representing strings, and `booleans`, with values `true` and `false`.

XML Schemata are translated into the intermediate representation to make it easier handling them, and finally into the type / assertion language of Ciao.

```
:- 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).
```

Fig. 4. Translation of types.

These type definitions are used to annotate the translated program and are eventually used by the analyzer. Figure 4 shows an actual translation, automatically obtained, 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 this elements is in turn a structure with two fields being an atom each.

We use a subset of XPath as the expression language. To simplify translation of expressions, we allow node navigation only along the descendant and attribute axes, to ensure that navigation is statically decidable and based only on structural typing.

4.3 Translation of Control Structures

The basic idea of the the translation is to keep track of the functional dependency of the resulting, response message on the input message with which a service is invoked. For every activity in the BPEL process, located at point i, a Prolog predicate $a_i(\bar{x},\omega)$ which performs a single action (corresponding to the activity) is generated. Point marker i is a symbolic indicator that is structured as a path from the root position to a node in the activity tree, which is needed for allow for branching and looping. Operation on point markers are: the next sibling activity i', and the k-th sub-activity i_k. $\bar{x}$ represents a collection of current values of all variables accessible at that point, and ω represents the state of the response message resulting from the execution from that point onwards, assuming it successfully terminates.

Depending on the type of activity it relates to, the predicate performs some condition testing, calculation, assignment or service invocation, and then directs the execution to the next activity by calling the predicate which will be in charge of implementing that activity, with an updated set of current variable values. Upon reception of the initial message, the accessible variables correspond to the input and the output messages, and when the end of the execution of the orchestration is reached the output message is out together and unified with the Prolog variable which is going to be returned to the caller. The same mechanism can be used to terminate process arbitrarily at any point. More formally, a basic activity A in position i is translated into

$$a_i(\bar{x},\omega) \leftarrow [\![A]\!](\bar{x},\bar{x}'),\ a_{i'}(\bar{x}',\omega)\,.$$

where $[\![A]\!]$ is the translation into Prolog of activity A which transforms the state $\bar{x}$ into the next state $\bar{x}'$, and $a_{i'}$ is the predicate corresponding to the next activity. For an assignment, $[\![A]\!]$ are the computation steps necessary to access the data (e.g., by unfolding or reconstructing data trees) or perform calculations as specified by the

XPath expression. For an invocation, $[\![A]\!]$ includes data access steps and the actual call to the predicate that models the invoked service.

At a some point i, the translation will reach the final of the chunk it is is charge of and there will not be a next activity in position i'. Then the translation is given with a clause of the form:

$$a_i(\bar{x}, \Omega(\bar{x})).$$

where $\Omega(\bar{x})$ denotes the term corresponding to the output message constructed with the current values of the reachable variables $\bar{x}$. This message is propagated upwards until it is returned to the top-level query or to some other translation of a BPEL process which calls the service being translated.

Structured activities have a different translation each. An if-then-else activity `if(`*Cond*`,` *Then*`,` *Else*`)` which is followed by an activity K is translated into two clauses:

$$a_i(\bar{x}, \omega) \leftarrow [\![Cond]\!], !, a_{i_1}(\bar{x}', \omega).$$
$$a_i(\bar{x}, \omega) \leftarrow a_{i_2}(\bar{x}, \omega).$$

where $[\![Cond]\!]$ succeeds iff *Cond* is met, predicate a_{i_1} corresponds to the translation of $(Then, K)$, and predicate a_{i_2} corresponds to the translation $(Else, K)$.

A `while` loop `while(`*Cond*`,` *Body*`)`, followed by K is translated likewise:

$$a_i(\bar{x}, \omega) \leftarrow [\![Cond]\!], !, a_{i_1}(\bar{x}', \omega).$$
$$a_i(\bar{x}, \omega) \leftarrow a_{i_2}(\bar{x}, \omega).$$

where a_{i_1} corresponds to the continuation $(Body, \alpha)$ in which α is translated as a recursive call back to $a_i(\bar{x}, \omega)$, and point i_2 corresponds to K. Other looping constructs can be derived from the scheme for the `while`: for instance, `repeatUntil(`*Cond*`,` *Body*`)` is converted into `(`*Body*`,` `while(`$\neg$*Cond*`,` *Body*`))`.

A new scope represented as `scope(`*VarDeclarations*`,` *Activities*`)` introduces new variables and may have associated fault handlers. These new variables change the structure of the environment in which the body of the scope is to be executed. Therefore, a state representation $\bar{z}$, different from $\bar{x}$, is needed. The translation is:

$$a_i(\bar{x}, \omega) \leftarrow a_{i_1}(\bar{z}, \omega), !.$$
$$a_i(\bar{x}, \omega) \leftarrow a_{i_2}(\bar{z}, \omega), !.$$
$$\ldots$$
$$a_i(\bar{x}, \omega) \leftarrow a_{i_n}(\bar{z}, \omega), .$$

where point i' corresponds to K, i_1 corresponds to $(Body, \beta, K)$ of the scope, and points $i_2..i_n$ correspond to fault continuations of the form $(Handler, \beta, K)$. In all continuations, translation of β adapts the output state representation $\bar{z}'$ at the end of execution of the scope's body/fault handler to a representation $\bar{x}'$ which is fed to the translation of K.

Raising an exception, which can eventually be caught by a fault handlers, is translated into the failure of the corresponding predicate. Since we are not interested in

exactly emulating the operational semantics of a BPEL process, in the translated code we do not record the identity of the exception, but rather assume that the correct fault handler is (non-deterministically) selected. That assumption enables the analyzer to calculate lower bounds for the computation in the case when no exception raised, as well as the upper bound taking into account the upper bounds for all handlers. Fault handler generation can be disabled in our translation, which is useful to analyze both the cases where there is no fault or where there is some exception has been raised.

Flows are modeled using the usual BPEL semantics, but without operationally paralleling the execution. Activities declared as `floats` within a `flow` have their join conditions and outgoing links among attributes, and support suppression of join failure in case when the input condition is not met. Activities are ordered to respect the dependencies. Links are internally declared as boolean variables visible in the static scope of the flow.

4.4 Handling Structured Data and State

The translation presented before needs to be made more sophisticated in order to take into account complex XML documents which are actually used as messages and as contents of BPEL variables. It is often the case that elements of XML documents are accessed (using XPath expressions), retrieved, and checked in, for example, `if-then-else` or `while` constructions. Modeling in Prolog the access primitives of XPath and including calls to these primitives directly in the translated code is of course a possibility which has the drawback that it almost certainly will confuse the analyzers: they will not be able to tell whether two equivalent accesses, performed in different points in code, actually point to the same field and would retrieve the same element.

To assist the analyzer in tracking component values and correlating the changes made to them, we take the approach of statically decomposing XML variables into the necessary components and carry them around explicitly as predicate arguments from that point onwards. Then, we no longer need to pass XML variables along with its components, because it can be reconstructed on demand. That transformation certainly makes the translated code a more difficult to understand for a human, but on the other hand it is not designed to be human-readable (although it is to a great extend), but to be amenable to analysis.

An inverse operation to unfolding a data tree is pruning it, which becomes necessary when a previously unfolded node receives a new value, as a result of an assignment or a reception of a response message from an invoked service. In that case, all of the descendant nodes in the data tree have to be invalidated and re-extracted from the new data through a process of unfolding. The ultimate reason is to keep the invariant that the values in the parameter list are always an up-to-date cache of elements in the XML tree.

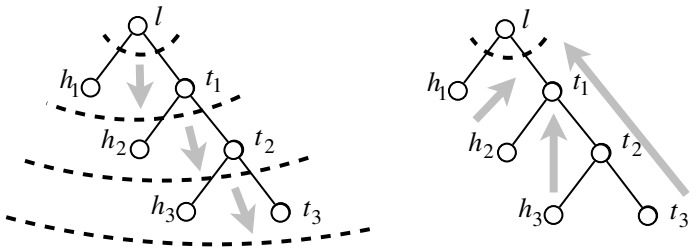

Fig. 5. Unfolding (left) and pruning (right) data tree of a list.

4.5 Analyzing Unavailable Code

One possible drawback of our proposal is the assumption of the availability of the BPEL code for remote services. There are good reasons for this unavailability: for example, because some provider does not want to reveal which code is being run in its servers. However, this turns out not to be a problem if we take into account that the analysis, while starting with BPEL code, does not actually act on it directly, but on an abstraction thereof which hides many details. Providers may offer this abstract code in order for third parties to check the complexity the providers claim. By doing so they would increment the confidence of their clients without revealing more than strictly necesssary.

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 such orchestration in our internal representation form instead of plain BPEL, as shown on 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 people to book a room 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 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 that would be impractical to present in full in this paper due to its size. Figure 5 provides some highlights on it. 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 translation of the main `while` loop, and the second clause finishes the process by constructing the answer from the current

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

```
:- 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',
      '$resp.body/hotres:personCount' <-
          '$req.body/hotres:personCount - 1' ]) ]) ].
```

Fig. 6. Abstract representation of a group booking process

value of the response variable. Part (c) shows the translation of service invocation, with previous unfolding of the outgoing message, and subsequent pruning of the response variable data tree.

```
:- 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.

Fig. 7. Translation into logic program

The resource analysis finds out how many times some specific operations will be called during the execution of the process. The resources we are interest in in this example are: the number of all basic activities performed (assignments, external

Resource	With fault handling		Without fault handling	
	lower bound	upper bound	lower bound	upper bound
Basic activities	2	$7 \times n$	$5 \times n + 2$	$5 \times n + 2$
Single reservations	0	n	n	n
Cancellations	0	$n - 1$	0	0

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

Table 3. Resource analysis results for the group reservation service

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. The results are displayed in Table 3, where the estimated upper and lower bounds are expressed as the 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).

6 Conclusions and Future Work

We have presented a resource analysis for BPEL which is based on a translation to an intermediate programming language (Prolog) for which complexity analyzers are available. These analyzers can be customized to analyze user-defined resources, thereby opening the possibility of generating resource-consumption functions, some of them of more interest for SOC than time complexity.

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 not affected. However, the behavior of the translation does not match, in general, that of the original process. 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 being worked on.

References

1. 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)

2. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, version 1.1. BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems. (2003)
3. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: 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)
4. 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)
5. 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)
6. Papazoglou, M.: Web Services: Principles and Technology. Prentice Hall (2007)
7. Cardoso, J.: Quality of Service and Semantic Composition of Workflows. PhD thesis, University of Georgia. (2002)
8. 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)
9. Cardoso, J.: Complexity analysis of BPEL web processes. Software Process: Improvement and Practice **12**(1), pp. 35–49 (2007)
10. 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)
11. 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'07). (August 2007)
12. 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
13. 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)
14. 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)
15. 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)