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

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

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

Abstract. Several activities in service oriented computing can benefit from knowing ahead of time future properties of a given service composition. In this paper we focus on how statically inferred computational cost functions on input data, which represent safe upper and lower bounds, can be used to predict some QoS-related values at runtime. In our approach, BPEL processes are translated into an intermediate language which is in turn converted 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. Actual and predicted time characteristics are used to perform predictive monitoring. A validation is performed through simulation.

Keywords: Service Orchestrations, Resource Analysis, Data-Awareness, Monitoring.

1 Introduction

Service Oriented Computing (SOC) is a well-established paradigm which aims at expressing and exploiting the computational possibilities of remotely interacting loosely coupled systems that expose themselves using service interfaces, while the implementation is completely hidden. Several services can be *put together* to accomplish more complex tasks through *service compositions*, written using some general-purpose programming language or a specifically designed language [1,2,3], and are usually exposed as full-fledged services.

* 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 have also been suported by projects FET IST-231620 *HATS*, Spanish FIT-340005-2007-14 *ES_PASS* (within EU ITEA2 06042-ESPASS) and 2008-05624/TIN *DOVES* projects, and CAM project S-2009/TIC/1465 *PROMETIDOS*.

A. Dan, F. Gittler, and F. Toumani (Eds.): ICSOC/ServiceWave 2009, LNCS 6275, pp. 414–424, 2010.
© Springer-Verlag Berlin Heidelberg 2010

SOC systems are expected to be active during long periods of time and operate across geographical and administrative boundaries. This requires monitoring and adaptation capabilities: monitoring compares the actual and expected behavior of the system (e.g., its QoS), and may trigger an adaptation if needed. Comparing the actual and the expected QoS of a composition —even assuming the composition is static— is far from trivial. *Predictive* monitoring aims at detecting deviations ahead of time by e.g. forecasting the future behavior. This is more complex but also more interesting and useful, as it can perform *prevention* instead of *healing*. Clearly, greater accuracy in calculating the expected QoS leads to better predictions.

Two factors, at least, have to be considered when estimating QoS behavior: the structure of the composition itself, i.e., what it does with incoming requests and which other services it invokes and how, and the variations on the environment, such as network links going down or external services not meeting the expected deadlines.

Of these two sources of information, the latter has been extensively studied [4,5,6,7], while the former has been less deeply explored. The actual data a service manages have been recognized as relevant [8,9], and actual message contents can greatly influence the runtime behavior of a composition (Section 2), but it has not been adequately addressed so far: prediction techniques which do not take run-time parameters into account are potentially inaccurate.

In this paper we will focus on applying a methodology, based on previous experience on automatic complexity analysis [10,15,16], which can generate correct approximations of complexity functions via translation to an intermediate language (Sections 3). These functions use (abstractions of) incoming messages in order to derive safe upper and lower bounds which depend on the input data and which are potentially more accurate than data-unaware approximations. In Section 4 we show how these functions can be used by monitoring to make better decisions and in Section 5 we make an experimental evaluation.

Correct data-aware cost functions can be useful for any situation where a more informed QoS estimation is an advantage. In particular, QoS-driven service composition [11,12,13] can use them to select better service providers for an expected kind of request, and adaptation mechanisms can also benefit from that knowledge [14].[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. *In a hotel reservation system (Fig. 1) a Client contacts a Booking Agency to request N hotel rooms. The Booking Agency uses a composed service that either books N rooms from a pool of K hotels, one at a time, or replies*

[1] The adaptation technique presented in [14] uses the same technique presented in this paper to derive cost bound functions. However, here we focus on the different problem of its application to predictive monitoring.

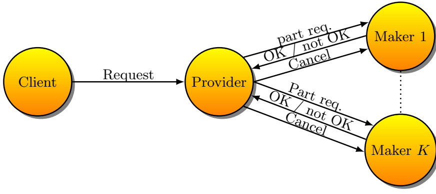

Fig. 1. Simplified hotel reservation system

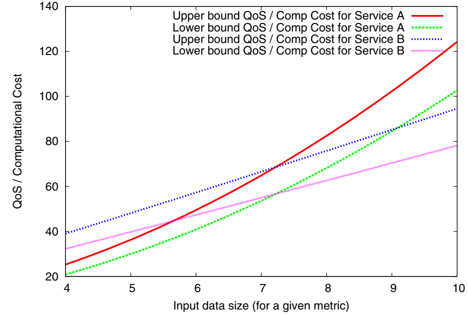

Fig. 2. Upper, lower bounds for two services

that no rooms are available. If not enough rooms are available after scanning all the hotels, it cancels any previously made reservation. A hotel without available rooms is excluded from further search. One message is used to for each room query, confirmation / rejection, and cancellation.

The whole process cannot be done in a single transaction, because the reservation systems of different hotels are 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: message exchanges can carry a sizable overhead, thus affecting actual execution time, or it is possible that the hotel reservation / booking service take a toll on every message they answer.

Assuming $K \geq N$, the *smallest* number of exchanged messages for a successful reservation $2N$ (N requests and replies to the same hotel) while the *greatest* number of messages is $2K + 3(N - 1)$ for the worst case of an unsuccessful reservation ($N - 1$ successful reservations, plus one last unsuccessful reservation which triggers their cancellation). Between these extremes, the maximum for a successful reservation would be $2K+2(N-1)$ messages. The complexity analysis depends both on the structure of the composition and on the values of N and K, which are composition parameters, since it is more likely that the hotels are listed in a separate registry, than hardwired into the composition code.

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

- If data is not taken into account, the impact of loops and conditionals can be only statistically estimated. *Guarantees* cannot easily be provided, as the value for any QoS attribute will be constant regardless of the actual values for K and N.
- Safe (upper and lower) bounds cannot usually be obtained, as probabilistic formulations usually rely on some kind of average.
- In QoS-aware matchmaking / rebinding, comparing different service compositions ignores the functional dependencies of QoS on the data.

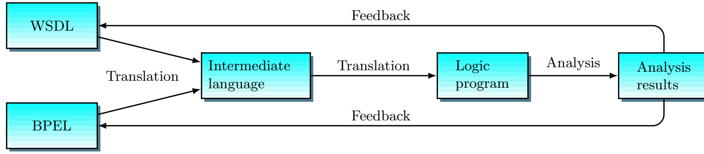

Fig. 3. The overall process

Table 1. Abstract orchestration elements

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

Figure 2 portrays upper and lower bounds of two compositions for some QoS as a function of a single input parameter. For some ranges of data input one composition is preferable over the other, while in the central zone we cannot decide.

3 Resource Analysis for Orchestrations

Due to space constraints, we present here an abridged version of the resource analysis technique in [14]. Our approach is based on translating process definitions into a language for which automatic computational cost analysis tools are available (see Fig. 3).

3.1 Overview of the Translation

Our orchestration language is a subset of BPEL 2.0 with WSDL meta-information. These are translated into an intermediate language which is then translated into the *Ciao* logic programming language [17]. The resulting logic program is then analyzed by the CiaoPP tool [18], which is able to infer upper and lower bounds for computational costs [15], among other analyses.

A BPEL process definition is translated into a service definition which associates a port/operation with the orchestration body. Processes forming a service network are translated into predicates that call each other to mimic service invocations. The intermediate language can describe XML schema-derived data types for messages, service ports, as well as relevant properties of external services of interest for the analysis when such services cannot be analyzed. Supported operations include generic constructs (assignment, sequence, loop...), and specific workflow constructs, such as flows, scopes, and invocations.

The translation does not follow strictly the operational semantics of the orchestration language: it just captures enough of it to ensure that the analyzers will infer correct information while minimizing precision loss.

Our analysis is currently restricted to orchestrations that start after receiving an initial message and finish by returning a reply or a fault notification. Stateful service callbacks using correlations are not supported. We support however a variant of the `scope` construct, which introduces local variables and fault / compensation handlers. We do not fully support compensation handlers, which in BPEL "undo" the effects of a scope using snapshots of variables recorded at successful completion of the scope. Except for recording snapshots, compensation handlers can be treated as pseudo-subroutines on a scope level, and inlined at their invocation place.

3.2 A Sketch of the Translation

The simple types in XML schemata are abstracted as three disjoint types: `numbers`, strings (translated into `atoms`), and `booleans`. Complex XML types are translated into predicates specifying how the type is built. The accepted expression language is a subset of XPath, so that navigation is statically decidable and components of XML structures can be passed as separate arguments when necessary to improve analyzer accuracy.

A process that implements operation o on port p is translated into a clause:

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

where X and Y correspond to the initial message and the final reply, and η is an environment that maps orchestration variables to logical variables. T is the translation operator, and $[A]$ is the process body. T is defined for both simple and structured activities, and may generate auxiliary predicates if needed. Table 2 sketches the translation of some activities.

A translation example is presented in Fig. 4. Subfigure (a) is a BPEL fragment of an orchestration that cancels the list of reservations and reports a fault, (b) is the corresponding intermediate form, and (c) is the translation into a logic program.

The resource analysis finds out how many times external service invocations will be performed during process execution, from which deducing the number of messages exchanged is easy. The upper and lower bounds estimates (as functions of input data) for the complete orchestration are displayed in Table 3. Results are given for both the fault-free execution case, and the more general case with possibility of faults.

Table 2. Translation of different activities

A	Translation of $T([A \mid R], \eta, V)$	A	Translation of $T([A \mid R], \eta, V)$
`reply`(v)	$V = \texttt{reply}(\eta(v))$ (*End orchestration*)	A_j, A_k	$T([A_j, A_k \mid R], \eta, V)$ (*Sequence*)
`throw`(f)	$V = \texttt{fault}(f)$ (*No fault handler*) $T([H], \eta, V)$ (*Insert fault handler*)	v `<-`e	$a(\eta, Y) \leftarrow E(e, \eta, X),\ T(R, \eta[X/v], Y)$ (*E evaluates e in env. η*)
`if`(c, A', A'')	$a(\eta, Y) \leftarrow C(c, \eta),\ !,\ T([A' \mid R], \eta, Y)$ $a(\eta, Y) \leftarrow T([A'' \mid R], \eta, Y)$	`while`(c, A')	$a(\eta, Y) \leftarrow C(c, \eta),\ !,\ T([A', A], \eta, Y)$ $a(\eta, Y) \leftarrow T(R, \eta, Y)$

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

(a) A BPEL code fragment

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

(b) The intermediate representation.

```
% ($i,$p,$resp.body/factory:part,$r,Y)
a_13(A,B,C,D,E):- A > 0, !, a_14(A,B,C,D,E).
a_13(A,B,C,D,E):- E =
  fault('factory->unableToComplete').
a_14(A,B,C,D,E):- F is A-1, a_15(F,B,C,D,E).
a_15(A,B,C,D,E):- nth(A,C,F), a_16(A,F,C,D,E).
a_16(A,B,C,D,E):-
  'factory->sales->cancelReservation'(B,F),
  (F=fault(G) -> E=fault(G)
  ;F=reply(H) -> a_13(A,B,C,H,E)).
```

(c) Translation into logic program.

Fig. 4. Translation example

Table 3. Resource analysis results for the group reservation service

Resource ($n \geq 0$: input arg. value)	**With fault handling** lower bound	upper bound	**Without fault handling** 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

4 Cost Functions for Monitoring

In this section we will describe how the expected value of some QoS characteristics can be derived from the value of resource consumption functions and the (expected) value of some environment characteristics, and we will also show how the availability of cost functions can be used to perform predictive monitoring.

4.1 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, a cost function cannot in general convey by itself 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

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

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

For example, in the case of the execution time of a single process, $cost_X(n)$ can be the number of activities executed, $\langle env_L, env_U \rangle$ the minimum / maximum time a single activity can take and $\oplus$ would be just multiplication. Since $\langle cost_L(n), cost_U(n) \rangle$ are lower and upper bounds, if $\langle env_L, env_U \rangle$ are also safe lower and upper bounds, the calculated QoS bounds will be safe lower and upper bounds of the actual (runtime) QoS values.

This generic scheme can admit variations: for example, a more accurate approximation can be constructed by assigning different weights to activities so that env_X is an array with a component for the execution time for every type of activity, $cost_X(n)$ is an array counting how many times every type of activity is executed, and $\oplus$ is the vector dot product.

4.2 QoS and Cost Functions during Composition Execution

In the general case, the cost function of a composition is made up of several parts related to different blocks of the composition structure. As an example, the upper bound of an `if-then-else` activity is the upper bound of the condition plus the maximum of the upper bounds of the `then` and `else` branches.

We can associate to every point in the composition a measure of how much resources "remain to be spent". In the `if-then-else` example, once the `if` part is over, what remains is the maximum of the upper bounds of the `then` and the `else` parts. This value depends on the point in the service composition it is measured and also on the values of the data at that moment. In a loop, where the same activity is executed several times, less "mileage" 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 "remaining" QoS: for example, from the activities still to be executed and the expected time of every activity, the time remaining for the completion of the composition can be derived. Measuring this remaining time is relevant from a monitoring point of view. Assuming that we have a faithful predictor of the QoS remaining until the end of the execution (such as that given by resource consumption functions and environment conditions), 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 with the actual environment conditions.

Figure 5 exemplifies such a situation. Let us assume we are interested in some QoS metric of a composition, whose value must not exceed *Max*. The cost functions and environment characteristics representing safe upper and lower bounds can be used here: if the **upper** bound is smaller than *Max*, then we have the guarantee that we do not violate the QoS boundary. If the **lower** bound is

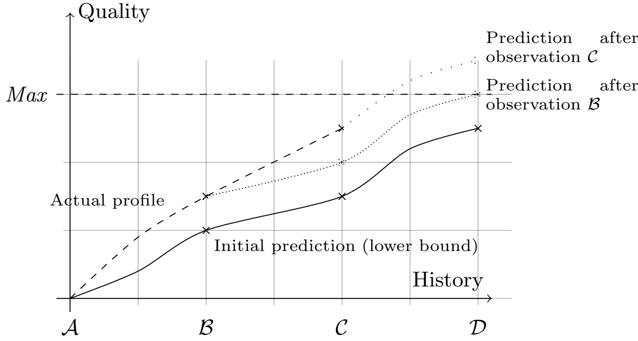

Fig. 5. Actual and predicted QoS throughout history

larger than *Max*, then we have the guarantee that we will violate the expected QoS. If none of these hold, then we cannot say anything for sure.

We designate four points ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$) in the execution of some composition and we will focus on how monitoring at these points can be predictively done with the use of cost functions. In Figure 5 the solid line represents the QoS initially predicted with the statically inferred cost functions and the expected environment conditions, while the dashed line represents the observed QoS.

At point $\mathcal{B}$, 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 can only be due to a change in the environment behavior (e.g., additional load on a server or a faulty network). An updated prediction for the future can be done by using the environment influence observed so far and the existing cost function. This new prediction curve (densely dotted) still ends, at point $\mathcal{D}$, within the limits of the acceptable range *Max*. However, at point $\mathcal{C}$ a new observation gives yet higher values for the QoS attribute. Yet another function and associated plot curve (sparsely dotted) can be constructed which predicts that the execution will violate the expected QoS. Therefore, at point $\mathcal{C}$ we have detected a problem before it appears and we can raise an alarm and maybe trigger an adaptation procedure. 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.

5 Experimental Evaluation

To validate the applicability of the proposed approach to predictive monitoring, we conducted a series of experiments which simulate the behavior of a service composition which may violate some QoS attribute (time, in this case) and we try to detect this situation ahead of time using analytically derived complexity bounds and the observed environment conditions. In our scenario, service

composition A is initiated with an input message, whose size (using some appropriate metric) is represented as an integer n (ranging, in this case, between 1 and 50). Upon message reception, service A iteratively invokes a partner service B and waits for a reply. The number of iterations is bounded by upper and lower bounds $\langle E_{AU}(n), E_{AL}(n)\rangle$ which grow linearly and which range between 100 and 500 and between 50 and 250, respectively, as n goes from 1 to 50.

Each invocation of B uses some time to transmit the invocation and its reply. Time is modeled as an environmental factor with bounds $\langle u_{AL}, u_{AU}\rangle$ that may change over time. Executing B involves a number of operations (or steps) independent from n with bounds $\langle E_{BL}, E_{BU}\rangle = \langle 8, 16\rangle$. Each step takes some time between the time-varying bounds $\langle u_{BL}, u_{BT}\rangle$. The initial values for the environmental factors (measured in milliseconds) are $u_{AL} = 7.5$, $u_{AH} = 20$, $u_{BL} = 3.75$, and $u_{BH} = 10$. The experiment is run under several regimes that differ on how environmental factors evolve.

The experiment assumes that the service composition A has to finish in at most $T_{\max} = 25,000$ ms. Violations to this requirement are detected by monitoring with the help of the complexity functions and the environment bounds, as previously described. The monitor periodically builds a *running* upper / lower bound estimate of the remaining execution time based on the elapsed time and the complexity / environmental factor bounds, respectively. The monitor issues a warning (Warn) when the upper bound of the estimate for the end-time of the task exceeds $T_{\max}$, to flag the risk of the time constraint violation, and an alarm (Alarm) if the lower bound estimate exceeds $T_{\max}$ to indicate that a violation of the time constraint is imminent under the current conditions. Alarm prevails over Warn.

We performed one hundred simulations for every value of n by randomly choosing, for each n, a concrete complexity value between the bounds for A and B. We measure the effectiveness of the approach by empirically assessing the frequency of violation given a warning/alarm status. Figure 6 shows alarm / warning / violation profiles for two environmental regimes. The one on the left simulates a system reconfiguration where environmental characteristics remain at their initial values until time $T_{max}/3$, when their bounds suddenly double (i.e., delays increase). The second regime (right) simulates a gradual degradation of

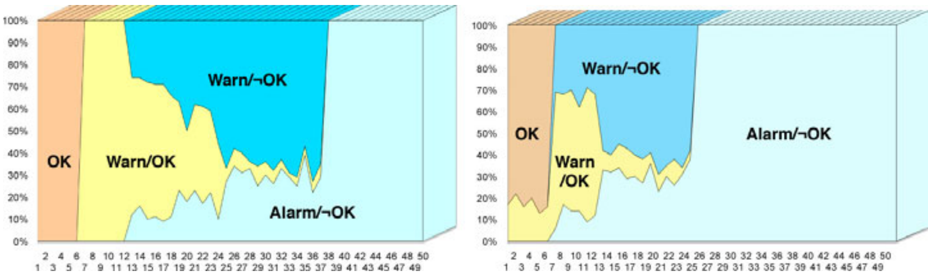

Fig. 6. Ratio of true and false positives for two environmental regimes

the system, where environmental bounds linearly increase over the period $T_{\max}$ by a factor of four.

Under the first regime, composition executions for small values of n take little time to complete, so they comply with the time limit (marked by OK) and no alerts are raised. For slightly larger input sizes (e.g. $n = 9$), executions still comply with the time limit, but warnings are raised (Warn/OK), since the monitor's estimate of the upper bound running time exceeds $T_{\max}$. As n increases, the number of false warning positives decreases in favor of the true warning positives (Warn/¬OK), because the average running time increases and thus the possibility of execution being affected by sudden deterioration of the environment factors. In the same region (around $n = 20$) some warnings start to be promoted to alarms, as the lower bound time estimates increasingly start to overshoot $T_{\max}$. These cases are marked with Alarm/¬OK, and they are all true positives, since the system degrades monotonically (things never get better). Further increases in n (to around 30) lead to rapid disappearance of the false warning positives. After $n = 38$, all executions fall into Alarm/¬OK, because the monitor is always able to detect ahead of time that the lower execution time bound overshoots the time limit.

In the second regime in Figure 6, featuring a gradual degradation, the upper execution time bound overshoots $T_{\max}$ in some cases even for very small input sizes (e.g. $n = 3$). A warning is then raised although no actual violations happen (executions are OK). As n increases, a pattern similar to that in the first regime is followed. For large values of n (but before the point in which it happened in the previous regime) all alarms rightly correspond to the $\neg OK$ case.

6 Concluding Remarks

We have sketched a resource analysis for service orchestrations based on a translation to an intermediate programming language for which complexity analyzers are available. The translation process 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 presented 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. We have applied them to perform predictive monitoring and the approach has been validated with a simulation which detects when the complexity bounds and the actual (simulated) execution cross the deadline and extracts statistical data regarding the accuracy of the predictions.

References

1. Jordan, D., et al.: Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, et. al (2007)
2. Zaha, J.M., Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Let's Dance: A Language for Service Behavior Modeling. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 145–162. Springer, Heidelberg (2006)

3. van der Aalst, W., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: The Role of Business Processes in Service Oriented Architectures. Dagstuhl Seminar Proceedings, vol. 06291 (2006)
4. Mukherjee, D., Jalote, P., Nanda, M.G.: Determining QoS of WS-BPEL Compositions. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 378–393. Springer, Heidelberg (2008)
5. Wu, J., Yang, F.: A Model-Driven Approach for QoS Prediction of BPEL Processes. In: ICSOC Workshops, pp. 131–140 (2006)
6. 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), 463–484 (2008)
7. Fugini, M.G., Pernici, B., Ramoni, F.: Quality Analysis of Composed Services through Fault Injection. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM Workshops 2007. LNCS, vol. 4928, pp. 245–256. Springer, Heidelberg (2008)
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), 35–49 (2007)
10. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems 15(5), 826–875 (1993)
11. Canfora, G., Penta, M.D., Esposito, R., Villani, M.: An Approach for QoS-Aware Service Composition Based on Genetic Algorithms. In: Proceedings of the 2005 conference on Genetic and Evolutionary Computation, pp. 1069–1075. ACM, New York (2005)
12. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)
13. Chen, Y.P., Li, Z.Z., Jin, Q.X., Wang, C.: Study on QoS Driven Web Services Composition. In: Zhou, X., Li, J., Shen, H.T., Kitsuregawa, M., Zhang, Y. (eds.) APWeb 2006. LNCS, vol. 3841, pp. 702–707. Springer, Heidelberg (2006)
14. Ivanović, D., Carro, M., Hermenegildo, M., López, P., Mera, E.: Towards Data-Aware Cost-Driven Adaptation for Service Orchestrations. Technical Report CLIP5/2009.1, Technical University of Madrid (UPM) (March 2010)
15. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-Definable Resource Bounds Analysis for Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007)
16. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 154–168. Springer, Heidelberg (2008)
17. 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: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 209–237. Springer, Heidelberg (2008)
18. 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), 115–140 (2005)