Determining QoS of WS-BPEL Compositions

Debdoot Mukherjee[1], Pankaj Jalote[2], and Mangala Gowri Nanda[1]

[1] IBM India Research Lab, New Delhi
{debdmukh,mgowri}@in.ibm.com
[2] Indian Institute of Technology, Delhi
jalote@cse.iitd.ac.in

Abstract. With a large number of web services offering the same functionality, the Quality of Service (QoS) rendered by a web service becomes a key differentiator. WS-BPEL has emerged as the de facto industry standard for composing web services. Thus, determining the QoS of a composite web service expressed in BPEL can be extremely beneficial. While there has been much work on QoS computation of structured workflows, there exists no tool to ascertain QoS for BPEL processes, which are semantically richer than conventional workflows. We propose a model for estimating three key QoS parameters - Response Time, Cost and Reliability - of an executable BPEL process from the QoS information of its partner services and certain control flow parameters. We have built a tool to compute QoS of a WS-BPEL process that accounts for most workflow patterns that may be expressed by standard WS-BPEL. Another feature of our QoS approach and the tool is that it allows a designer to explore the impact on QoS of using different software fault tolerance techniques like Recovery blocks, N-version programming etc., thereby provisioning QoS computation of mission critical applications that may employ these techniques to achieve high reliability and/or performance.

Keywords: Quality of Service, composite web services, workflows, BPEL.

1 Introduction

Services Oriented Architecture aims to provide infrastructure for a marketplace wherein more services will be produced by composing various web services rather than coding programs from scratch. As in any competitive market, where a number of offerings are available for the same functionality, Quality of Service is slated to be the key differentiator. Knowing the QoS of the web service being composed is extremely crucial during the process of service orchestration (binding concrete web services to tasks in the workflow). The integrator of the WS-composition has to judiciously choose every web service that he binds to the composition in order to attain a high level of QoS and meet his Service Level Agreement(SLA) requirements. A tool that can estimate the QoS of the resultant WS-composition, given the values of QoS parameters for constituent web services will come in most handy for the integrator. This paper aims to provide such a framework for QoS determination in WS-BPEL [1] processes.

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 378–393, 2008.

Business Process Execution Language (BPEL) has emerged as the de-facto standard for representation of industrial workflows. BPEL has been proven to be more expressive than traditional workflow modeling languages [2] - most of which support only block structured flow constructs. Some of the key features of BPEL that distinguish it from other workflow languages are: (i) *Synchronization links* and the *transition* and *join conditions* that one can impose on these links, (ii) *Fault handlers* and *compensation handlers* to support fault handling and backward recovery of long running transactions respectively, and (iii) Event driven programming constructs like *receive*, *pick* and *event handlers*.

Despite BPEL emerging as the standard for web service composition, most of the work on QoS of composite web services has so far focused on composition through structured workflows [3,4]. As mentioned, BPEL is semantically more powerful than workflow languages, hence QoS computation using conventional workflows can handle only a small subset of BPEL programs. In this paper, we determine *Reliability*, *Response Time* and *Cost* of any composition expressed in standard WS-BPEL. The model also allows adding fault tolerant constructs like *N-version programming* [5], *Recovery block* [6], *Return the Fastest Response* and *Deadline Mechanism* to enhance the reliability and performance of the tasks in the BPEL composition and then determining their impact on QoS. Although the use of fault tolerant constructs in web services have been studied in literature [7,8,9], there has been no research that quantifies the QoS improvement that may be brought about by these constructs.

Outline of our Approach. For determining QoS, the BPEL file is parsed to build an activity graph that consists of activities as nodes and captures the dependencies between activities that arise from the control flow structure of the BPEL process. All QoS computations happen at the level of a node in this graph. The dependencies of an activity are tracked in order to determine the probability with which the activity may start execution and also the time instant in a run of the BPEL process when it is fired. These estimates about an activity alongwith QoS estimates of its children help us to model its QoS. A recursive algorithm runs through the activity graph computing QoS parameters at each node and recombining these values to arrive at reliability, response time and cost for the overall WS-BPEL process. The entire approach has been implemented in a tool that takes as input a BPEL process and QoS of constituent web services and computes QoS for the composition. In this paper, we illustrate the approach by applying it to an E-Governance application that models the passport office workflow. Our tool also offers the flexibility to replace an invocation of a web service by a more robust invocation of a set of web services tied via fault tolerant constructs. QoS improvement achieved through such a design can then be precisely measured through rules defined specific to fault tolerant constructs that aggregate QoS of constituent web services. The major contributions of the paper include providing a QoS determination framework for WS-BPEL processes and provisioning an environment where integrators of composite web services can try out fault tolerant designs and gauge QoS of such processes at design time.

2 WS-BPEL (Business Process Execution Language)

WS-BPEL lays down a grammar for capturing the behavior of a business process based on interactions between the process and its partner processes. The root element of any WS-BPEL file is ⟨*process*⟩ - an element that outlines the scope of the business process and encloses declarations for all *partner links*, *variables*, *handlers* and an *activity*(this activity may in turn contain other activities). An activity can be of two types: *basic* or *structured*.

Basic activities either describe interactions with other partners or model primitive steps in the process. Basic activities include - ⟨*invoke*⟩ to call operations; ⟨*receive*⟩ and ⟨*reply*⟩ to accept and respond to inbound messages respectively; ⟨*assign*⟩ used to carry out updates on variables; ⟨*wait*⟩ to introduce delays; ⟨*exit*⟩ to immediately end the business process; ⟨*throw*⟩ and ⟨*rethrow*⟩ to signal internal faults; and ⟨*empty*⟩ to do nothing.

Structured activities encode control-flow logic and can have other activities nested in them. WS-BPEL enumerates seven different structured activities. A ⟨*sequence*⟩ contains one or more activities that are performed in the order in which they appear within the ⟨*sequence*⟩ element. A ⟨$flow$⟩ provisions execution of activities concurrently and also allows for synchronization between the activities contained in it through the notion of *links*. An ⟨if⟩ consists of one or more conditional branches defined by the ⟨if⟩ and optional ⟨$elseif$⟩ elements, followed by an optional ⟨*else*⟩ element. A ⟨*pick*⟩ waits for the occurrence of exactly one event from a set of events and executes the activity contained within that event. The events can either be receipt of inbound messages (⟨$onMessage$⟩) or triggering of timer based alarms (⟨$onAlarm$⟩). WS-BPEL 2.0 supports three forms of loop constructs - ⟨*while*⟩, ⟨*repeatUntil*⟩ (both checked by truth value of the ⟨*condition*⟩ set in them) and ⟨$forEach$⟩ (controlled by an implicit index variable that is initialized to ⟨$startCounterValue$⟩ and ends in ⟨$finalCounterValue$⟩).

WS-BPEL's notion of a ⟨*scope*⟩ offers the ability to specify a behavioral context within which an activity may execute. A scope allows definition of variables, partner links, message exchanges and correlation sets that are visible only within the scope. Event handlers, fault handlers, a compensation handler, and a termination handler may also be attached to a scope.

3 A Running Example - Passport Application Service

We present an example of a passport office workflow implemented with the help of WS-BPEL to elucidate the key concepts throughout this paper. The composite web service for the passport workflow calls upon operations in external web services to verify parameters such as age and date of birth whose validation is required for issuing a passport. Age may be verified by either an education board or a municipal office. After verification of age and address, one may proceed to issuing a passport only if the bank payment has been made by the applicant.

A part of the BPEL process is shown graphically in Figure 1. The synchronization links named X, Y and Z and the ⟨*transitionCondition*⟩s specified on

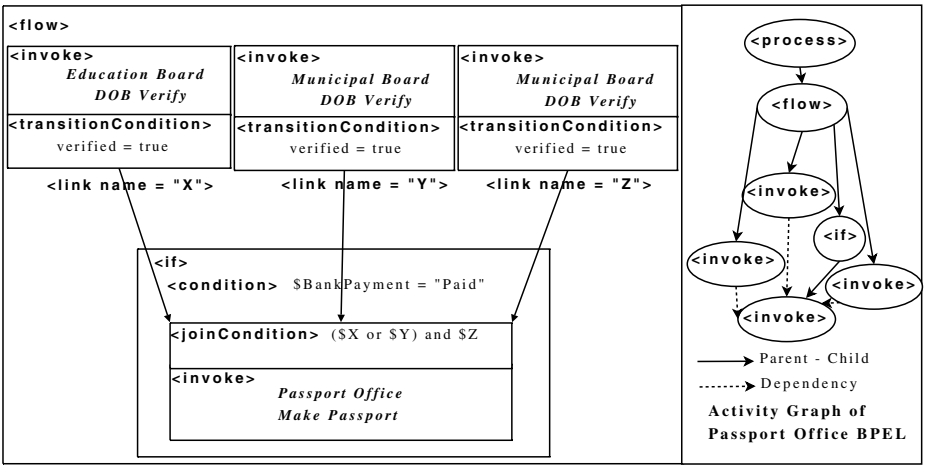

Fig. 1. Passport Office BPEL Process

them are used to ensure that the web service to issue passport may be started only if the furnished documents have been successfully validated. The boolean expression on the links in the $\langle joinCondition \rangle$ at the *invoke* for passport issue web service examines whether at least one age-proof is valid and the given address proof has been verified. Our QoS model will require as input reliability, time and cost for each of the four constituent web services and the probabilities of success of each of the transition conditions and the if condition.

4 Determining QoS

Our QoS model estimates reliability, response time and cost for a WS-BPEL process and is capable of dealing with the complex graph-like control flow structures, event driven programming constructs and fault handling mechanisms that may be present in it. Since activities in WS-BPEL may be heavily intertwined with synchronization links, one cannot perform reductions analogous to those proposed in [3] and derive QoS of a block by simply aggregating the QoS of its constituents. In order to infer QoS of an activity we track all the dependencies that the activity may have and obtain their QoS. An activity A is said to be *control dependent* on another activity B, if A may only start execution after the completion of B. In WS-BPEL, a control dependency of an activity A may be another activity B if B is either the source of an incoming link to A or B precedes A in a *sequence*.

The model captures the three key QoS parameters addressed by it in the following way:

– *Reliability* of a BPEL process is calculated as the probability that the process scope will successfully complete execution.

- *Response Time* is a random variable characterized by values for mean and standard deviation representing the expected time taken for completion of the process.
- *Cost* of a BPEL composition is calculated as the aggregate of expected costs of all activities contained in it, assuming that each web service invocation incurs some fixed cost.

Inputs to the Model. Apart from the values of reliability, response time and cost for all web service invocations of the BPEL process, the model also assumes certain parameters that characterize the control flow of the business process to be available as inputs. These include the probability of selecting branches or events in *if* and *pick* activities respectively, the average number of iterations in loops and for each *catch* or *catchAll* block the fraction of failures of its associated scope that it successfully intercepts. Average waiting times for all $\langle receive \rangle$ activities (that are not start activities) and all $\langle onMessage \rangle$ events that are used to intercept inbound messages to the business process are required by the model. All of these attributes can be determined from the execution log of the business process.

4.1 Overall Approach

All computations in our model occur at the level of an *activity* or a *scope* or a *handler*, which are various units of encapsulation of process logic in WS-BPEL. The BPEL workflow is represented as an *activity graph* where the activities/scopes/handlers are represented by nodes. A node, X, in the BPEL process is annotated by: (a) its child nodes, i.e., activities that are directly contained by X (b) its control dependencies. Additionally, nodes for *invoke* activities and *scopes* are annotated with *catch* blocks, *fault handlers*, *event handlers* and *compensation handlers* that they may be associated with. The node corresponding to the $\langle process \rangle$ scope forms the root node of this activity graph. Figure 1 shows the activity graph obtained out of the passport office BPEL process.

Basic Elements of the Model. QoS computation of a BPEL process requires that we compute for each node X in this graph - (a) $P(\mathcal{S}_X)$: Probability that X *successfully* completes execution in a single run of the process, (b) ET_X: Expected end time or the time of completion of X measured relative to the start of the process, (c) $Cost_X$: Sum of the expected costs of all its child nodes. These three parameters for the root node of the activity graph give reliability, response time and cost respectively for the WS-BPEL composition.

Successful completion of a node encompasses the corresponding activity / scope / handler delivering its desired functionality to the effect that it measures upto the expectations of all other nodes that might be dependent on it. In determining $P(\mathcal{S}_X)$ of a node X, we make use of $P(start_X)$ which is the probability that the activity may start execution in a state that is semantically in accordance with one which is expected at that point. $P(start_X)$ abstracts all effects that the dependencies of a node X may have on $P(\mathcal{S}_X)$. The computation of $P(\mathcal{S}_X)$ may involve use of $P(start_X)$ and a suitable aggregation (separately defined for

Table 1. Algorithm `QoS Computation`

Algorithm $setQoS(X)$: Set $P(\mathcal{S})$, ET and $Cost$ for a node X
for all Z such that Z is a dependency of X **do**
$setQoS(Z)$
end for
Compute $P(start_X)$, ST_X and PC_X
for all Z such that Z is a child of X **do**
$setQoS(Z)$
end for
Compute $P(\mathcal{S}_X)$, ET_X and $Cost_X$ according to rules specific to Type of X.

each activity type) of $P(\mathcal{S})$ of all activities / scopes / handlers contained in X. Response time computation involves obtaining for each activity their start and end times of execution in some run of the business process. ST_X denotes the time instant when all dependencies of X are complete and X is ready to start. The end times of all nodes nested within X are estimated and suitably aggregated with ST_X to obtain ET_X. $Cost$ is computed more on the lines of the reduction based approach. PC_X is the conditional probability that X will start execution in an instance of the BPEL process given that the parent of X starts execution in the same instance. $Cost_X$ for a node X is given by the sum of costs of all its child nodes relaxed by their PCs.

Algorithm Description. After the BPEL XML document has been parsed to prepare the *activity graph*, we can proceed with QoS determination. Algorithm 1 presents the recursive structure of QoS computation followed for any node in our activity graph. If we invoke the function $setQoS()$ on the root node, all nodes of the activity graph are traversed. $P(\mathcal{S})$, ET and $Cost$ are set in each of them before we obtain these parameters for the root node and hence QoS of the BPEL process. $P(\mathcal{S})$, ET of all dependencies of X are required to compute parameters $P(start_X)$, ST_X that help to abstract the effects of the dependencies of X in the QoS computation of the children of X. The termination of Algorithm 1 is guaranteed because well-formed WS-BPEL processes cannot have control cycles formed with the help of links (See SA00072 in [1]).

4.2 Determining $P(start)$, Start Time (ST) and PC

The methodologies to calculate $P(start_X)$, ST_X and PC_X for any node X remain the same irrespective of the type of X whereas determination of $P(\mathcal{S}_X)$, ET_X and $Cost_X$ happen according to rules defined specific to each activity type (See Section 4.3).

The computation of $P(start_X)$ requires the probability, $P(joinCondition_X = true)$, that the join condition (explicit or implicit) attached to the node X evaluates to true. A *join condition* is a boolean expression on the incoming links of a WS-BPEL activity and its status decides whether the activity can start execution. A *transition condition* on a link is defined at its source and refers

to the condition that must hold good for the link to attain a true value. The probability that a link A_i assumes a true value is dependent on the successful completion of its source activity and its transition condition being evaluated to true. In our model, $P(transitionCondition_{A_i} = true)$ is obtained as an input.

$$P(A_i = true) = P(\mathcal{S}_{source(A_i)}).P(transitionCondition_{A_i} = true) \quad (1)$$

Example: In our passport application example, the link X connects the *invoke* activity for *DOB Verify* to that for *Make Passport*. Thus, $P(X = true)$ or simply $P(X)$ may be computed as a product of $P(S_{invoke_{DobVerify}})$ and the input probability for success of the transition condition.

If we have an *AND* in the boolean expression of the join condition, we consider the intersection of events that the constituent links are true and in case of *OR* we determine the union of those events. To compute the probability that the join condition for an activity evaluates to true, $P(joinCondition_X = true)$, we convert the boolean expression into a canonical Sum of Products (SOP) form which is evaluated with the help of the standard law of probability for union of events, assuming that the events $P(A_i = true)$ for all links A_i are independent. It may be noted that in this work we only carry out a control flow analysis on WS-BPEL processes, if we track data flow too we can do away with the assumption of links being independent and achieve more accurate QoS estimates.

Example: We determine the probability $P((X \cup Y) \cap Z)$ for the join condition at the *Make Passport invoke* after we compute values of $P(X)$, $P(Y)$ and $P(Z)$.

A WS-BPEL activity, X, may start execution if the following conditions are met.

- The parent activity of X has started execution which guarantees that the control dependencies of the parent have completed execution.
- If X is a child of a $\langle sequence \rangle$, then the prior activity (if any) in the $\langle sequence \rangle$ has successfully completed execution.
- If X contains incoming synchronization links then its join condition (implicit/explicit) has been evaluated to true.

Thus, in order to compute $P(start_X)$, we take a product of the following probabilities, if they are applicable: (a) $P(start_{parent_X})$, (b) $P(\mathcal{S})$ of predecessor in *sequence*, (c) $P(joinCondition_X = true)$. Note, that we consider $P(start_{parent_X})$ only if X is one of the following - (i) first activity inside a *sequence*, (ii) activities inside a *flow* that do not have incoming links (iii) activities in all branches of *if* or *pick* (iv) activity inside a scope. The other children of a sequence / flow are directly or indirectly dependent on these first activities, so their dependency on their parent gets captured in our model. Loops are handled differently (See Section 4.3), and thus the calculation of $P(start)$ of a child activity of a loop does not consider $P(start_{loop})$.

Example: In case of the Make Passport WS invocation, we note that only (a) and (c) are applicable and $P(start_{invoke_{MakePassport}})$ is given by $P(start_{if})$. $P((X \cup Y) \cap Z)$.

The start time (ST) for an activity is taken to be the maximum of the end time of its predecessor in *sequence*, the expected end times of all the source activities of its incoming links and the start time of its parent. For the activities inside message based events in *pick* and the receive activities (that are not start activities, i.e., *createInstance* = *"no"*), we also add their average waiting times to their start times.

The cost model computes for each activity the probability that its starts given its parent has already started. This probability, referred to as PC, is used extensively in our model for aggregation of the costs of child activities in order to estimate the expected cost of an activity.

$$\begin{aligned} PC_X &= P(start_X | start_{parent(X)}) \\ &= \frac{P(start_X)}{P(start_{parent(X)})} \text{ Since, } P(start_{parent(X)} | start_X) = 1 \end{aligned} \tag{2}$$

4.3 Activity-Wise Rules for Determining $P(\mathcal{S})$, ET and $Cost$

In this section, we detail our formulations to estimate $P(\mathcal{S})$, ET and $Cost$ for every WS-BPEL activity. It may be noted that for any activity reliability modeling must be performed before determination of time and cost may take place.

Basic Activities. All *basic activities* except *invoke* have zero costs associated with them and are assumed to complete successfully and instantaneously when they start. Hence, the probability of successful completion, $P(\mathcal{S})$ of a basic activity is equal to the probability that it gets to execute, $P(start)$ and its end time(ET) is equal to its start time(ST). Only, ET of a *wait* activity is computed differently after adding the delay specified to its ST.

Invoke. *Invoke* activities denote the point of calling external web services. These may be prone to failures and have an associated latency and cost. For each invoke activity, the model expects as input the reliability (R_{ws}), response time (T_{ws}) and cost (C_{ws}) of the web service bound to it. R_{ws} represents the conditional probability R_{ws} that an invocation to an external web service fails despite the call being made with proper arguments. R_{ws} and T_{ws} incorporate failures and latencies respectively that arise both at the service site or from the network. Now, $P(start_{invoke})$ gives the probability that the invoke activity begins in a consistent state with proper arguments available. Thus, we may write:

$$P(\mathcal{S}'_{invoke}) = R_{ws} \times P(start_{invoke}) \tag{3}$$

$$ET'_{invoke} = ST_{invoke} + T_{ws} \tag{4}$$

$$Cost_{invoke} = C_{ws} \tag{5}$$

However, an *invoke* activity may have *catch* blocks and *compensation handlers* attached to it and completion of an *invoke* activity would encompass their completion too. Thus, we may write the expression for $P(\mathcal{S})$ and ET of an invoke activity after accounting for the attached catch blocks and compensation handler if there are any. QoS determination for handlers is discussed later in this section.

Structured Activities. QoS computation of structured activities require determination of QoS parameters of their children. We briefly describe here the rules for estimating $P(\mathcal{S})$, ET and $Cost$ for all structured activities. Table 2 lists the equations for QoS determination in structured activities.

Table 2. QoS of Structured Activities

Activity	$P(\mathcal{S})$	ET	$Cost$
Sequence	$P(\mathcal{S}_{lastChild})$	$ET_{lastChild}$	$\sum_i PC_{child_i} \times Cost_{child_i}$
Flow	$\prod_{\forall i} P(\mathcal{S}_{sink_i})$	$Max_{\forall i}(ET_{sink_i})$	$\sum_i PC_{child_i} \times Cost_{child_i}$
If	$\sum_i P(sel_i) \times P(\mathcal{S}_{br_i})$	$\sum_i P(sel_i) \times ET_{br_i}$	$\sum_i P(sel_i) \times PC_{br_i} \times Cost_{br_i}$
Pick	$\sum_i P(sel_i) \times P(\mathcal{S}_{evt_i})$	$\sum_i P(sel_i) \times ET_{evt_i}$	$\sum_i P(sel_i) \times PC_{evt_i} \times Cost_{evt_i}$
Loop	$P(start_i) \times P(\mathcal{S}_{child})^n$	$ST_i + n \times ET_{child}$	$n \times Cost_{child}$

Sequence. An activity nested inside a *sequence* can only start if the previous activity in the sequence has been successful. Thus, if the i^{th} child of sequence is executing, then all child activities from the first to the $(i-1)^{th}$ can be taken to be complete. Therefore, we can model $P(\mathcal{S})$ and end time of a *sequence* by that of its last child. The expected cost of a *sequence* is simply a weighted sum of the expected costs of all its child activities with their PCs being the weights.

Flow. A *flow* activity is deemed to complete only if all activities enclosed by it are complete. However, since the synchronization links in effect model the control flow of execution, it may be contended that the completion of all child activities without any outgoing links would mark the completion of the *flow*. Therefore, $P(\mathcal{S})$ and end time for a *flow* activity may be modeled as an aggregation of that of the *sinks* which are children with no outgoing links. Cost of a *flow* is modeled exactly the same way as that of a *sequence*.

Example: The *flow* in the passport office workflow has only one sink namely the *if* activity. Thus, both $P(\mathcal{S}_{flow})$ and ET_{flow} will be given by those for the *if* activity.

If. An *if* activity is complete when the activity nested in the taken branch completes. It completes immediately if no *condition* evaluates to true and no *else* branch is specified. All QoS parameters - $P(\mathcal{S})$, ET and $Cost$, of an *if* activity are calculated as weighted sums of the values of the same for the activities contained in all its branches, the weights being the probability, $P(sel_i)$,with which a branch gets selected for execution. Note that in cost modeling, the costs of the branch activities are relaxed by their PCs while aggregation.

Example: In the *If* activity in our example, $P(\mathcal{S}_{if})$ is computed by taking a weighted sum of that of the *invoke* for *Make Passport* and an *empty* activity ($P(\mathcal{S}_{empty})$ is always 1) assumed to be present in the non-existent else branch.

Pick. *Pick* activities are treated in a similar way as *if* activities; with probabilities of selection of each event being taken as input.

Loops. QoS modeling for the activity inside a loop may be performed independently without requiring QoS information of its parent activity or activities

outside the loop. This is facilitated by the WS-BPEL stipulation (See SA00070 in [1]) that synchronization links cannot enter into repeatable constructs by crossing their boundaries. The various iterations of the loop are assumed to be independent and in effect the loop construct is treated as a number of copies of the contained child activity running in sequence. In case of *while* and *repeatUntil*, the number of iterations, n, is obtained as an input to the model. In a *forEach* activity, the number of iterations, n, is obtained by parsing the values of $\langle finalCounterValue \rangle$, $\langle startCounterValue \rangle$ and the *completion condition* if specified. The special case where the loop is rolled in parallel (the *parallel* attribute being set to true) is handled by taking time taken by one iteration only whilst calculating response time. Again, PC of the child activity of a loop will always evaluate to 1 because it cannot have any dependencies, so there is no scope of relaxation of the cost of the child of a loop.

Handlers. The handlers in WS-BPEL impact QoS of *scopes* and *invoke* activities to which they may be attached. For lack of space, we present here only the QoS modeling for fault handlers. (Refer to [10] for a discussion on Compensation Handlers and Event Handlers)

The probability that fault handlers start execution is dependent upon the rate of faults thrown up by the web service invocation or scopes.

$$FaultRate_X = \begin{cases} 1 - R_{ws} & \text{if X is an invoke,} \\ 1 - P(\mathcal{S}_{scopeChild} | start_{scope}) & \text{if X is a scope} \end{cases} \tag{6}$$

The model assumes as input, the fraction of faults caught by each catch block $FractionCapture$, out of the total number of faults produced by its scope. Thus, the probability that a catch block starts may be given by:

$$P(start_{catch_i}) = FaultRate_X \times FractionCapture \times P(start_X) \quad \text{where X = scope/invoke} \tag{7}$$

$P(\mathcal{S})$, ET and $Cost$ of a catch block are taken to be the same as that of the activity nested in it. The fraction of faults removed FFR by all catch blocks in a fault handler may be computed as:

$$FractionFaultRemoval(FFR) = \sum_{\forall i} (FractionCapture_i \times P(\mathcal{S}_{catch_i})) \tag{8}$$

After taking into account the fraction of faults removed, the improved probability of the *invoke / scope* will stand as below:

$$P(\mathcal{S}''_X) = P(\mathcal{S}'_X) + FaultRate_X \times FFR \tag{9}$$

where, X may be *invoke*/*scope*.
Time and Cost of a fault handler (FH) may be given as:

$$T_{FH} = Max_{\forall i}(FractionCapture_i \times FaultRate \times (ET_{catch_i} - ST_{catch_i})) \tag{10}$$

$$Cost_{FH} = FaultRate \times \sum_{\forall i} FractionCapture_i \times Cost_{catchActivity_i} \tag{11}$$

Scope. The following equations model QoS of a scope in presence of fault handlers(FH), compensation handlers (CH) and event handlers (EH). Similar equations may be written for *invoke* to incorporate the effects of attached handlers.

$$P(\mathcal{S}_{scope}) = P(\mathcal{S}''_{scope} \times P(\mathcal{S}_{CH}) \times P(\mathcal{S}_{EH}) \quad (12)$$

$$ET_{scope} = Max(ET'_{child} + T_{FH}, ET_{CH}, ET_{EH}) \quad (13)$$

$$Cost_{scope} = Cost_{child} + Cost_{FH} + Cost_{CH} + Cost_{EH} \quad (14)$$

The above equations when applied to the $\langle process \rangle$ (that is nothing but a special form of scope) will give the QoS of the composite web service written in BPEL.

Suppression of Join Failures. The entire exposition above assumes that a $bpel : joinFailure$ is thrown whenever a join condition is not satisfied. However, if *suppressJoinFailure* attribute is set to *yes*, failure of a join condition results in the activity being skipped and a false status being propagated on all its outgoing links with no fault generation. Since a skipped activity is also deemed to be complete, the expression for the probability of successful completion is improved by the probability of the activity being skipped.

$$P(\mathcal{S}') = P(joinCondition = true) \times P(\mathcal{S}) + (1 - P(joinCondition = true)) \quad (15)$$

However, in the evaluation of $P(link = true)$ (See Equation 1), the probability of successful completion of the source will substituted by old $P(\mathcal{S})$ and not $P(\mathcal{S}')$ because a failed status is propagated on each of the outgoing links of a skipped activity.

5 Impact of Fault Tolerance on QoS Computation

During orchestration of a web service composition, the designer may find that all web services available to perform some task do not meet reliability requirements or show huge variations in their response times. In such cases, fault tolerant constructs may be used to create dependable web services out of undependable ones and attain the desired reliability and performance levels. Our QoS model and the tool supports four conventional fault tolerance (FT) techniques - *N-version programming*, *Recovery Blocks*, *Return Fastest Response* and *Deadline Mechanisms* - and helps in QoS determination after application of these FT-constructs. The first two constructs focus on reliability improvement and the latter two seek to enhance performance. Again for lack of space, we only derive expressions for QoS computation of the *Deadline Mechanism* construct in this paper (Details of the other 3 constructs can be found in [10]). The following discussion assumes for every task in the workflow we have a set of web services $(A_1, A_2, \ldots, A_n)$, having diverse designs, for which QoS parameters - reliability (R), response time (T) and cost (C) are known.

Deadline mechanism supports setting deadlines for completion of tasks and provision forking off redundant services if a primary service does not complete

within some specified length of time. In our model, the user sets a hard deadline, HD, for completion of a task. Moreover, the designer specifies for each alternate web service A_i, the time instant TF_{A_i} when it may be fired if no response is received from services that have been running. In any invocation of a *Deadline Mechanism (DM)* block, the service that returns first gets counted. For each service in a DM block, we find the probability that it returns the first response and use these probabilities to compute the reliability and time of a DM block.

$$R_{DM} = \sum_{i=0}^{n} P(T'_{A_i} = Min\{T'_{A_1}, T'_{A_2}, \ldots, T'_{A_n}\} \text{ and } T'_{A_i} < HD) \times R_{A_i} \quad (16)$$

$$\text{where, } T'_{A_i} = T_{A_i} + TF_{A_i}$$

$$T_{DM} = Min_{\forall i}\{T'_{A_i}\} \quad (17)$$

Again, a service inside a DM block may start only if a successful response has not been generated by other services till the point of its firing. Thus,

$$P(Start_{A_i}) = P(TF_{A_i} \geq Min\{T'_{A_1}, T'_{A_2}, \ldots, T'_{A_n}, HD\}) \quad (18)$$

Cost of a DM block is a weighted sum of the costs of all services in it, the weights being the $P(Start)$ of the services, i.e.,$\sum_{i=0}^{n} P(Start_{A_i}) \times C_{A_i}$.

6 Implementation

We have implemented our QoS model for WS-BPEL 2.0 processes in a standalone software using Java 1.5. The tool accepts as input a BPEL source and parses it to ask the user for the various control flow parameters (See Inputs to the Model in Section 4) and the values for reliability, time and cost for each web service invocation present in it. The tool outputs the three QoS parameters for the overall process and also shows $P(\mathcal{S})$, ET and $Cost$ for every activity. For improving QoS through fault tolerant constructs the user has to provide WSDLs as well as QoS values of the redundant web services and the web services to be used for voting and assertion checking. Figure 2 shows the block diagram of our implementation and Figure 3 shows some of the user interfaces that are a part of the tool.

7 Related Work

Cardoso's thesis [3] is the seminal work in literature to propose a framework for estimation of QoS in web service processes. He proposes Stochastic Workflow Reduction (SWR) to arrive at QoS estimates for the overall workflow, provided the QoS values for all tasks in the workflow are known. The SWR algorithm repeatedly applies a reduction on various structured constructs until only one atomic task remains. He introduces reduction rules for sequential, parallel, conditional, loop and fault tolerant systems. However, there is a restrictive rider

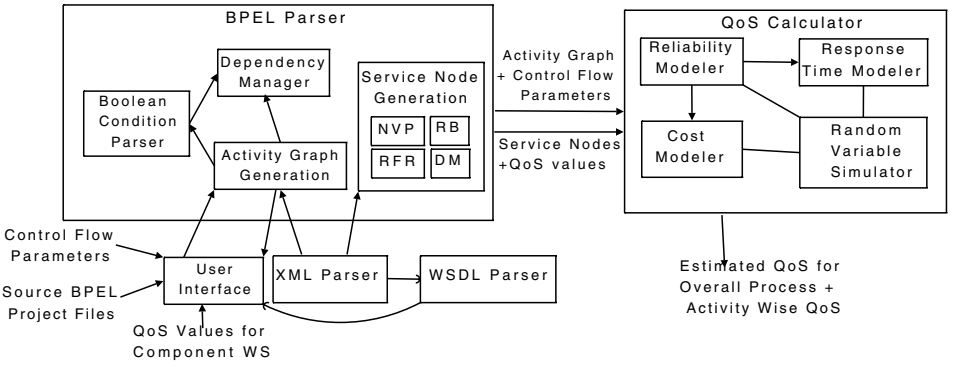

Fig. 2. Block Diagram of Implementation

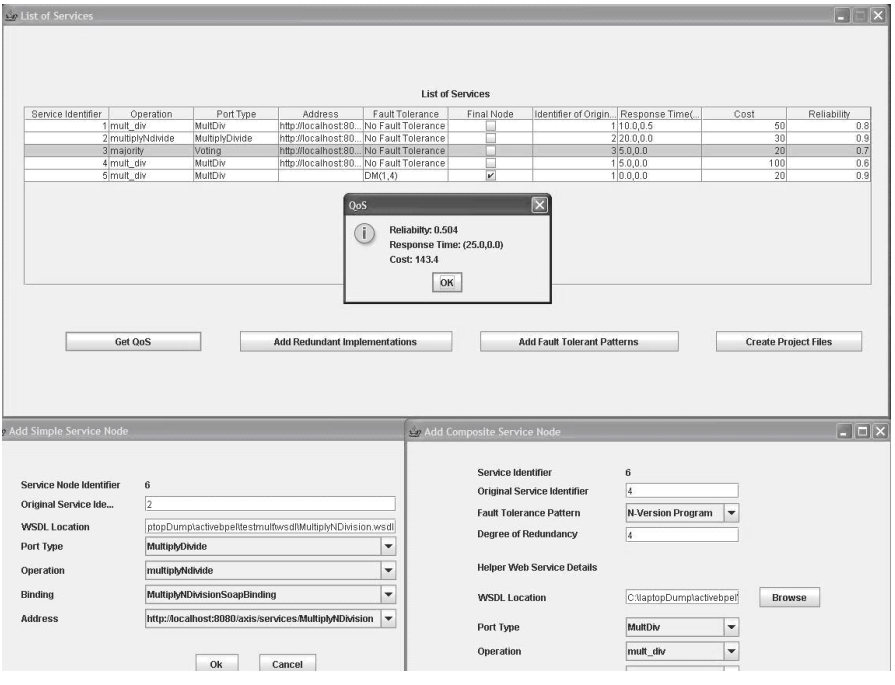

Fig. 3. User Interfaces in QoS tool

added with most of the reduction rule-sets given above. For example, in the sequential system the start task cannot be a split and the end task cannot be a join. Cardoso's model adapts from the reductions used in standard reliability theory for computing reliability of series-parallel systems [11][12]. But the model is not capable of handling complex systems such as the one shown in Figure 4(a) that can neither be reduced to a series nor decomposed as a parallel system.

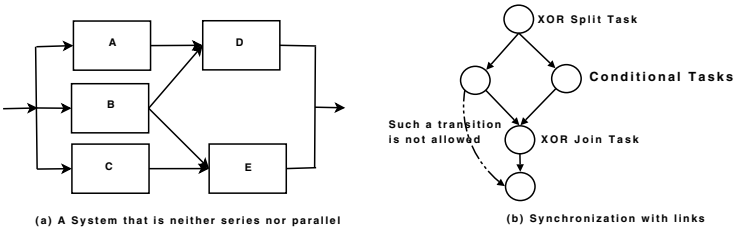

Fig. 4. Limitations of Structured Workflows

Again, presence of goto-like transitions that extend from one structured construct to another as shown in Figure 4(b) prevent application of the proposed reductions.

Hwang et. al. [13] propose a probabilistic framework for QoS computation. They extend Cardoso's model to have each QoS parameter for a web service represented by a discrete random variable having a certain probability mass function (PMF). Canfora et al. [14] apply Cardoso's QoS model with minor modifications in their middleware that uses genetic algorithms for QoS aware composition and replanning. Zeng et al. [4] model composite web services as state charts and put forward aggregation functions to ascertain QoS of *execution plans*. Jaeger et al. [15] propose aggregation of QoS dimensions on the workflow patterns listed in Van der Aalst's seminal work [16]. The approach is an elegant one but the authors do not explain how to dig out such workflow patterns from a process an to carry out an implementation of the same. D'Ambrogio and Bocciarelli [17] propose a model driven approach wherein a BPEL process is described by an UML model which is then annotated with performance data to obtain a LQN (Layered Queueing Network) model that may be solved to predict performance. Although the process of conversion of models built according to the UML Profile into BPEL has been thoroughly described in [18], the complex control flow offered by BPEL have not been exhaustively mapped back onto the UML profile. However, BPEL to UML transformation is an active research topic [19].

8 Conclusions

We have presented a comprehensive QoS determination model for WS-BPEL processes. Our QoS calculator provides values for reliability, response time and cost for each activity / scope / handler. WS-BPEL is capable of expressing arbitrarily complex structures with the help of synchronization links between activities and facilitates fault handling, event driven programming and backward recovery through compensation handlers. Although QoS research in workflows has gained importance, no QoS estimation technique exists in literature to the best of our knowledge, that captures graph-like control flow structures or supports the kind of behavior that is rendered through WS-BPEL handlers. Our QoS tool enables the designer to modify parts of the BPEL process workflow

(differently organize the control flow or add/remove fault handlers and compensation handlers etc.), add fault tolerance constructs to it and check for QoS improvement.

The limitations of the model are: (i) It does not support detection of mutual exclusiveness of links at a join making estimation of $P(start)$ less accurate in face of mutually exclusive paths. (ii) It does not handle *isolated scopes* that provide control over concurrent access of shared resources. (iii) Forced termination behavior is not captured by the model and it does not deal with termination handlers. All of these issues lay scope for future work in the area.

References

1. OASIS WS-BPEL Technical Committee, Web Services Business Process Execution Language Version 2.0 (2007)
2. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of Web Services Composition Languages: The Case of BPEL4WS. In: Proceedings of the 22nd International Conference on Conceptual Modeling, ER (2003)
3. Jorge, A., Cardoso, S.: Quality of Service and Semantic Composition of Workflows, Ph.D. Thesis, University of Georgia, Athens, Georgia (2002)
4. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions in Software Engineering (2004)
5. Avizienis, A., Chen, L.: On the implementation of N-version programming for software fault tolerance during execution. In: Proceedings of IEEE COMPSAC (1977)
6. Randell, B.: System structure for software fault tolerance. In: Proceedings of the international conference on Reliable software (1975)
7. Dobson, G.: Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 126–133 (2006)
8. Gorbenko, A., Kharchenko, V., Popov, P., Romanovsky, A., Boyarchuk, A.: Development of Dependable Web Services out of Undependable Web Components, School of Computing Science, University of Newcastle (2004)
9. Looker, N.M., Xu, M.J.: Increasing Web Service Dependability Through Consensus Voting. In: Computer Software and Applications Conference (2005)
10. Mukherjee, D.: QoS in WS-BPEL Processes, M.Tech. Thesis, Department of Computer Science & Engineering, Indian Institute of Technology, Delhi (2008)
11. Grant Ireson, W., Coombs Jr., C.F., Moss, R.Y.: Handbook of Reliability Engineering and Management. McGraw Hill, New York (1996)
12. Hoyland, A., Rausand, M.: System Reliability Theory: Models and Statistical Methods. John Wiley and Sons, Chichester (1994)
13. Hwang, S.-Y., Wang, H., Tang, J., Srivastava, J.: A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. Journal of Information Sciences (2007)
14. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of the 2005 conference on Genetic and evolutionary computation (2005)

15. Jaeger, M.C., Rojec-Goldmann, G., Muhl, G.: QoS Aggregation for Web Service Composition using Workflow Patterns. In: Proceedings of the Enterprise Distributed Object Computing Conference (2004)
16. Van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Journal of Distributed Parallel Databases (2003)
17. D'Ambrogio, A., Bocciarelli, P.: A model-driven approach to describe and predict the performance of composite services. In: Proceedings of the 6th international workshop on Software and performance (2007)
18. Amsden, J., Gardner, T., Griffin, C., Iyengar, S.: Draft UML 1.4 profile for automated business processes with a mapping to BPEL 1.0 (2004)
19. Reiter, T.: Transformation of Web Service Specification Languages into UML Activity Diagrams, Diploma thesis, University of South Australia (2005)