

Dependability Assessment of Web Service Orchestrations

Salvatore Distefano, Carlo Ghezzi, Sam Guinea, Raffaella Mirandola

Abstract

In this paper, we focus on the reliability and availability analysis of Web service (WS) compositions, orchestrated via the Business Process Execution Language (BPEL). Starting from the failure profiles of the services being composed, which take into account multiple possible failure modes, latent errors, and propagation effects, and from a BPEL process description, we provide an analytical technique for evaluating the composite process' reliability-availability metrics. This technique also takes into account BPEL's advanced composition features, including fault, compensation, termination, and event handling. The method is a design-time aid that can help users and third party providers reason, in the early stages of development, and in particular during WS selection, about a process' reliability and availability. A non-trivial case study in the area of travel management is used to illustrate the applicability and effectiveness of the proposed approach.

Index Terms

Web service, business process execution language, fault propagation, fault compensation and termination, event handling.



ACRONYMS AND ABBREVIATIONS

BPEL Business Process Execution Language

CDF Cumulative Distribution Function

DTMC Discrete Time Markov Chain

-
- *The authors are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy. E-mail: {salvatore.distefano,carlo.ghezzi,sam.guinea,raffaella.mirandola}@polimi.it*

FCT Fault, Compensation and Termination

HTTP HyperText Transfer Protocol

QoS Quality of Service

rv Random Variable

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

UDDI Universal Description Discovery and Integration

WS Web Service

WSDL Web Services Description Language

NOTATION

γ^X response probability of activity X (γ^{X^U} user, γ^{X^A} architect)

A_C^{WS} WS container's availability

α_S^{WS} WS inner service response probability

n_l number of latent error inputs or outputs

n_f number of faulty inputs or outputs

n number of total inputs or outputs ($n = 1 + n_l + n_f$)

\mathbf{P}^X propagation matrix of activity X (\mathbf{P}^{X^U} user, \mathbf{P}^{X^A} architect, architect γ -propagation matrix $\mathbf{P}^{X^{A\gamma}}$)

\mathbf{P}_{cl}^X correct-latent error block propagation matrix of activity X

\mathbf{P}_f^X fault block propagation matrix of activity X

\mathbf{In}^X input vector of activity X (\mathbf{In}^{X^U} user, \mathbf{In}^{X^A} architect)

$\overline{\mathbf{In}}^X$ normalized input vector of activity X on the n_l+1 correct-latent error inputs

\mathbf{O}^X output vector of activity X

$\overline{\mathbf{O}}^X$ normalized output vector of activity X on the n_l+1 correct-latent error inputs $\overline{\mathbf{O}}^X = \{(\overline{\mathbf{In}}^X \cdot \mathbf{P}_{cl}^X)/(p_{ok}^X), \mathbf{0}\}$

\mathbf{I}_k k -order identity matrix

p_{ok}^X probability of a non-faulty output of x , $p_{ok} = \sum_{j=0}^{n_l} \mathbf{O}^X[j]$

C^X correctness, probability that X produces a correct output given it was correctly invoked

E_i^X error probability that an execution of X with correct input returns an error mode $1 < i \leq n_l$ as output

G_j^X fault probability that a correct invocation of X triggers the $1 < j \leq n_l + 1$ fault

B_i^X error robustness, probability that X , invoked with error mode $0 < i \leq n_l$ as input, masks the error and returns a correct output

SE_i^X error susceptibility, probability that X , invoked with error mode $0 < i \leq n_l$ as input, produces a latent error output

SF_i^X fault susceptibility, probability that X , invoked with error mode $0 < i \leq n_l$ as input, produces a fault output

1 INTRODUCTION AND MOTIVATIONS

Service orientation has played an extremely important role in the evolution of Information Technology in the last decade. It provides the foundations for some of today's most significant advancements, such as Web 2.0, Cloud computing, and the Internet of Things. The service abstraction imposes that we rethink methods and techniques for developing and managing both physical (i.e. hardware infrastructure) and logical systems (i.e. software architecture). In the former case, we speak of *Service Oriented Infrastructure*, while in the latter we speak of *Service Oriented Architecture*.

The term Service Oriented Architecture (SOA) refers to an *ecosystem* [1] of interacting processes, physical nodes, and people that create, manage, and provide functionalities as services. According to this perspective, a *business process* is a complex service that combines simpler, loosely coupled, reusable Web services (WSs) using *service orchestration*. Service orchestrations are often implemented using *workflow languages* that provide mechanisms for selecting and composing services through the definition of complex control- and data-flows. The Web Services Business Process Execution Language (BPEL) is the de-facto standard workflow language [2] for SOA.

In this paper, we focus on service reliability and availability, and discuss how they can be properly addressed at design-time when composing services with the BPEL language. The ability to perform an early assessment of these qualities, instead of waiting for the implementation and runtime stages, is a key challenge for SOA architectures, and a key factor when implementing dependable software.

In SOA, WSs are owned by different providers, and used as black boxes. To support service selection, providers expose WS properties in specific Web-accessible *registries*. Selection is then implemented as a query to the registry that returns the services that are known to match a specific set of requirements. This approach allows us to choose among different alternatives, as long as the providers explicitly specify QoS information about their WSs (e.g., reliability, availability, and performance).

However, even if QoS information is available, how the orchestration will be carried out cannot be entirely foreseen at design time. It will depend on many different aspects, such as the availability of the involved services, how the services respond, the status of the network, unexpected error conditions, etc. Non-functional properties are a crucial concern in composite process orchestration, and architectural decisions, such as WS selection and workflow structuring, always affect the QoS of the resulting process.

We advocate that, thanks to SOA, designers can reason on BPEL process reliability and availability at a high level of abstraction. We expect well-founded methods to be available to compute whether the non-functional aspects of the WSs we include in a composition satisfy the reliability and availability requirements we have for the process.

We have investigated failures in component based systems in previous work [3], [4]. In this work, we considered multiple failure modes, as well as the emergence, propagation, and transformation of errors in a running system's data and control flows, and how these can eventually lead to a failure.

BPEL processes can be affected by many different failure modes, because they orchestrate and aggregate services that come from different providers. They mix multi-tier and heterogeneous domains, as well as different approaches and technologies, through the use of interoperable interfaces. The aim of this paper is, therefore, to provide a technique for evaluating the reliability and availability of composite services, with a complete coverage of the BPEL language. We want to understand reliability, availability, response probability, and propagation phenomena. To gain this understanding, we take into account the service parameter values and the definition of the process, with its fault, compensation, termination (FCT), and event handling. To reach this goal, we have revised and extended our previous work in light of the SOA application domain, and we have identified, and specified, both the phenomena that we want to observe, and

the parameters that allow us to quantify them.

Several approaches have been developed in the past to model and evaluate the reliability and availability of composite processes (as we discuss in Section 2). However, to the best of our knowledge, none of them adequately takes into account FCT and event handling, which are typical of BPEL composite processes, and none of them considers these aspects in conjunction with propagation phenomena. Furthermore, our proposed technique is lightweight and scalable. The number of computational operations grows linearly with the number of BPEL process activities. This feature will enable us in the future to use the technique to predict reliability and availability anomalies and violations, and to effectively support service providers in decision making.

2 RELATED WORK

Many different aspects of Web-based systems have been studied over the last few years, such as workload characterization, performance, availability, and reliability. Our work proposes an approach to analyze the reliability and availability of Web service compositions, orchestrated via the BPEL workflow language. We base our method on the software reliability engineering approach [5], and propose an early design-time reliability assessment, to prevent late fault discovery. In particular, we provide an analytical evaluation that takes into account multiple possible failure modes, latent errors, and propagation effects.

Recently, with the emergence of self-adaptive architectures, several approaches have been proposed in literature to deal with self-healing business processes [6], [7]. Although our approach is mainly focused on design-time evaluation, its extension to support run-time self adaptation is currently being investigated.

In our previous work [3], [4], we addressed the stochastic evaluation of reliability in component-based systems, and we considered multiple failure modes and failure propagation phenomena. Other related work is presented hereafter, classified in three main areas, described next.

Web-based availability and reliability. Discrete time Markov chains (DTMC) have often been used in literature [8], [9], [10] as suitable models for system analysis. A hierarchical approach for e-business systems has been proposed in [8]; [9] presented a

model of an e-commerce site in which user navigation patterns were represented; in [10] Web user activity was modeled as an on-off process combined with a Markov process. An empirical analysis of Web system availability from the end-user's perspective is presented in [11].

Related work on Web services' availability and reliability includes several papers with analytical models (e.g., [9], [12]), and empirical studies (e.g. [13]). The analytical models exploit different kinds of Markov processes to define availability and reliability models for a composite Web service. The empirical analyses consider both the workloads and the reliability of Web servers, and distinguish between inter-session and intra-session Web characteristics [13]. More recently, some papers have tackled the problem of composing a service-oriented system from publicly available Web services (e.g., [14]), taking into account different types of Web service failures.

Architecture-based reliability. Architecture-based software reliability analysis has been dealt with in several papers, and specific surveys on this topic can be found in [15], [16]. These papers mainly focus on evaluating the overall system reliability by taking into account the internal failure of each component, and the probabilities of their interactions. Interesting empirical studies, and works that deal with uncertainty analysis of architecture-based software reliability, can be found in [17], and [18]. Among the existing works in the area, the ones that mostly influenced our work are briefly summarized below. In 1980, Cheung [19] proposed the so called *user-oriented reliability* approach, and defined it as the probability to observe a correct output from a program, given a representative set of input data from the external environment. Here, system reliability is derived using a stochastic Markov process that describes the system as a set of interacting components, expressed as a function of the component reliability, and of utilization. In the area of self-assembly service-oriented computing, Grassi [21] proposed an approach for automatic reliability estimation that exploits the compositional aspects that are inherent in these applications, and their dependency on external sources. Error propagation among components is completely neglected in these works.

The work of [22] is one of the first papers in this area. It proposes a graph theory-based reduction approach for the evaluation of software's non-functional properties, such as its reliability and performance. Later, several other applications of this technique were

proposed in different contexts (DTMC [23], workflow [24], [25], etc.). [24], [25] also dealt with WS and business processes adopting the reduction technique; however, they do not consider different failure modes and corresponding propagation aspects.

Error propagation. In [27], error propagation probability is defined as the probability that an error, after manifesting itself, will propagate through components, and possibly end up visible at the user interface level. This definition is limited to a single type of failure, and in [27] it is supported by a methodology and a tool capable of analyzing the sensibility of each component, with respect to failure and error propagation. Different approaches based on fault injection have been applied to estimate error propagation in software systems during the testing phase (e.g., [28]).

3 OVERVIEW OF THE APPROACH

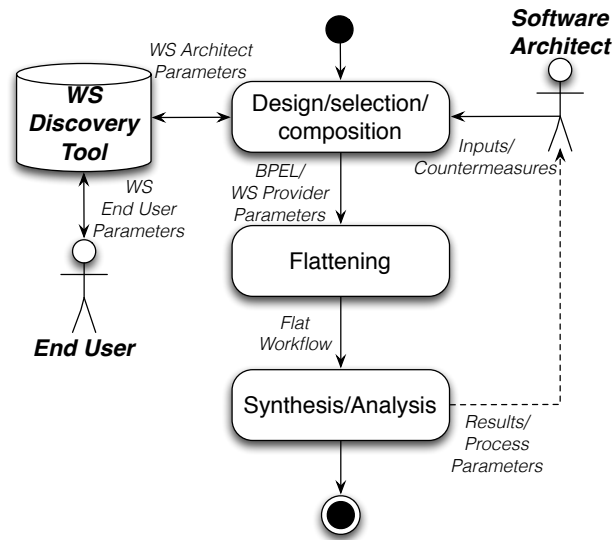


Fig. 1. The processing schema

The aim of this work is to evaluate the overall reliability and availability parameters of a composite business process described in BPEL [2], knowing its internal control- and data-flows, and the non-functional properties of its participating services.

As shown in Fig. 1, these systems have two main kinds of stakeholders, each with its own interests: the *end user*, and the *software architect*. From the user's point of view,

the goal is to *select* the service that best fits the requirements; from the architect's point of view, the goal is to *implement* the best possible process by composing the available services. Specifically, the differences in the users' and architect's objectives impose that the technique be flexible with respect to the parameters it will consider. This flexibility strongly depends on how and what is considered as correct, or faulty, for each WS interaction. For example, starting from the taxonomy of [29], WS faults or errors can be managed internally by a process, and returned to the user as valid outputs, or they can be directly forwarded to the user as errors.

We propose to organize the process evaluation into three steps, as shown in Fig. 1.

i) *Design-selection-composition* - In this step, a composite service is designed by orchestrating external WSs. External Web services are selected for composition via specific discovery tools (e.g. *Universal Description Discovery and Integration* registries, WS portals, or WS search engines), according to their functional and non-functional properties and to how we expect them to contribute to the properties of the composition. The process design is usually performed by a software architect, who must be aware of the semantics of the functional and non-functional parameters to be considered. Because the technique we propose provides an analytical solution, it is necessary to specify an adequate formulation of the problem, and to define appropriate metrics and parameters, as discussed in Section 4.

ii) *Flattening* - The BPEL process includes nested subprocesses or activities, and manages faults and events through specific FCT and event handlers. To deal with this issue, the hierarchically structured BPEL process, and the non-functional properties of its services and internal activities have to be transformed into a *flat workflow*. Unlike the BPEL process, the flat workflow only has one termination. This step is required and is non-trivial, as we will discuss in Section 5.

iii) *Synthesis and Analysis* - The overall non-functional properties of the BPEL process are evaluated in terms of the properties of the individual services that are orchestrated by the workflow. Aggregation-reduction rules and sensitivity analyses are applied, as described in Section 6.

The results can be fed back to the software architect, who can then modify the original process, either using the same services or by selecting new ones. If the results are

satisfactory, they can be published to a service registry and made available for further selection and composition. The analysis technique can be used both at design and run time, because it has a low complexity¹. A way to use the technique could be to identify the WSs that have a large impact on the process' reliability metrics, using sensitivity and importance analysis, as explained in the example discussed in Section 7, or to infer the properties of one or more components given the requirements on the whole process.

4 WS SELECTION AND COMPOSITION: THE ANALYTICAL FRAMEWORK

To evaluate the reliability and availability of a composite BPEL process, we need to take into account a detailed view of the overall software architecture. Fig. 2a illustrates the four main server components involved in the deployment and execution of a BPEL process, while Fig. 2b highlights the architecture from both the end user's and the architect's perspectives.

The *Web service* is the basic building block of a service-oriented system. It is the smallest composable unit, and therefore should be designed to maximize reusability. The *SOAP engine* is responsible for de-serializing incoming requests, for providing them to a service instance, and for serializing outgoing responses (e.g., Apache Axis). In the case of composed WSs, the SOAP engine is usually part of a more sophisticated execution environment called the *BPEL engine*, a centralized environment for executing and managing composite processes. Typically, the SOAP and BPEL engines are part of an *application server* (e.g. Jakarta Tomcat) that provides a place to hold applications that must be accessed by different clients. Some application servers already include hypertext transfer protocol (HTTP) functionality; otherwise we also need a *Web server*, which is a software capable of handling HTTP messages (e.g., Apache HTTP Server).

End users are not capable of distinguishing between correct versus faulty service responses and correct versus faulty BPEL or SOAP WS responses; their view of the WS stack is composed of solely three layers: the HTTP server, the application server, and the BPEL and SOAP engine-service. On the other hand, software architects are aware of the composite service's intended logic, and know how to distinguish service

1. It scales (sub-)linearly with respect to the number of the process elements-; see Section 6.3

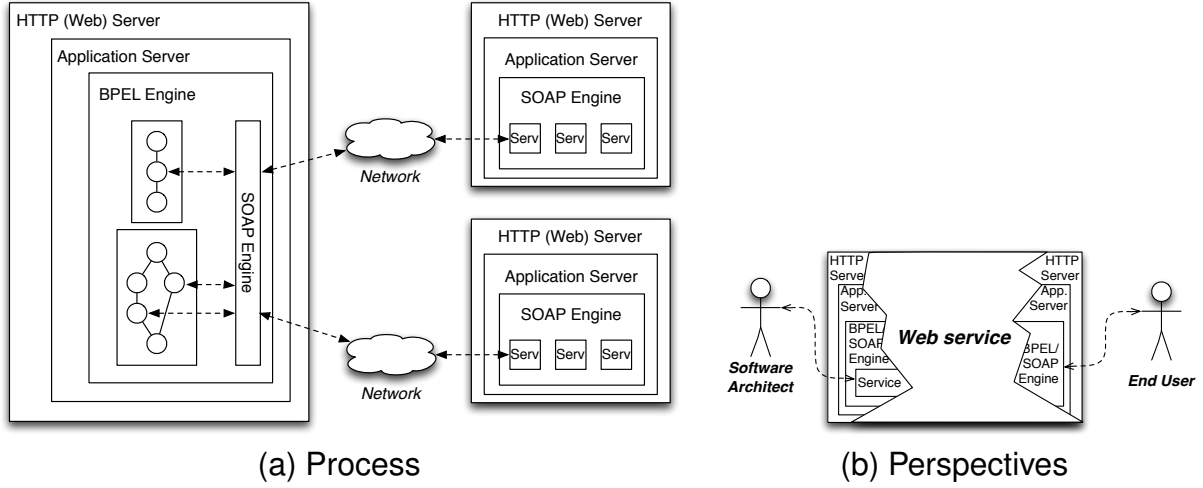


Fig. 2. WS stack, process WS composition (a), and response's perspectives (b).

responses from engine responses; therefore, they are interested in composing processes that provide adequate responses when invoked. Thus, from the architect's perspective, the full WS stack needs to be considered.

A WS system can be formally described as follows.

Definition 1: A WS system WS is characterized by the pair

$$\langle \gamma^{WS}, \mathbf{P}^{WS} \rangle$$

where

- $\gamma^{WS} \in [0, 1] \subset \mathbb{R}$ is the *WS response probability*, i.e. the probability the WS system will provide a response; and
- $\mathbf{P}^{WS} \in [0, 1]^n \subset \mathbb{R}^n \times [0, 1]^n \subset \mathbb{R}^n$ is the *propagation matrix*, including the probabilities that the input of WS is propagated to the output, where n is the number of total possible (correct, latent error, or fault) inputs and outputs.

We can further characterize the previous definition according to the two identified perspectives, obtaining $\langle \gamma^{WS^U}, \mathbf{P}^{WS^U} \rangle$ in the case of end users (U), and $\langle \gamma^{WS^A}, \mathbf{P}^{WS^A} \rangle$ for architects (A).

A similar formalization is adopted for BPEL's internal activities, thus representing an internal activity Act by a pair $\langle \gamma^{Act}, \mathbf{P}^{Act} \rangle$.

It is important to understand these parameters in terms of the architecture of Fig. 1.

Response probability: *Response probability* can be defined as

$$\gamma^{WS} = Pr\{\text{The whole WS system provides a response}\}.$$

According to Fig. 2a, a WS system can be decomposed into two parts: the WS container (HTTP Server, Application server, BPEL and SOAP engines), and the service itself. The former is in charge of forwarding the request to the latter, which in turn performs the actual processing, and returns a result. Because the container does not actually process the request, we are mainly interested in its *readiness*, i.e. the capability to forward messages to and from the service. The service, on the other hand, has to elaborate the request, and therefore we have to quantify its capability to *operate*.

Avizienis et al. [29] define *availability* as the readiness for correct service, and *reliability* as the continuity of correct service. Following this approach, we can identify and characterize the WS container's availability A_C^{WS} as its readiness to provide access to the actual service. This is why, when we calculate the WS response probability, we need to consider the *cumulative distribution functions* (CDF) of its inner service's i) time to failure $F_S^{WS}(\cdot)$ (or reliability $R_S^{WS}(\cdot) = 1 - F_S^{WS}(\cdot)$), and ii) time to response $T_S^{WS}(\cdot)$. Assuming the container and the inner service time to failure are statistically independent, we can express the WS response probability as

$$\gamma^{WS} = Pr\{\text{The WS container is available, **and** the inner service provides a response}\}.$$

There are three reasons why a WS container can become unavailable: i) it can suffer from HTTP (Web) server failures and unavailability, ii) the Application Server may not be able to dispatch service requests, or iii) there may be errors in the BPEL-SOAP Engine. In the first two cases, the system generates and delivers an HTTP error message (e.g. 400, 404, 500, 502, 503) to the user. In the third case, a specific SOAP fault message is delivered to the user. In composed WSs, inter-component communication problems are detected by the innermost engine, i.e. the SOAP engine. That is why they are considered errors in the BPEL-SOAP engine.

The service container's availability A_C^{WS} is therefore the probability that the HTTP

server, the Application server, and the BPEL-SOAP engines are working correctly:

$$A_C^{WS}(t) = Pr\{\text{The servers and the engines are working at } t\}.$$

Assuming that availability reaches a steady state after a transient, we can characterize these quantities using single values, i.e. the steady state availability $A_C^{WS} = \lim_{t \rightarrow \infty} A_C^{WS}(t)$. More specifically, if A_{HTTP}^{WS} , A_{AS}^{WS} , and $A_{BPEL-SOAP}^{WS}$ are the steady state availabilities of the HTTP (Web) server, the Application server, and the BPEL-SOAP engine, respectively, the container service availability is $A_C^{WS} = A_{HTTP}^{WS} A_{AS}^{WS} A_{BPEL-SOAP}^{WS}$.

If the WS container is available, the WS's response probability will depend on the inner service actually providing the response, i.e. its response probability is strongly related to the inner service's operation continuity or reliability. If Q is the inner service's lifetime, then

$$R_S^{WS}(q) = Pr\{\text{The service is working in } [0, q]\} = Pr\{Q > q\} = 1 - F_S^{WS}(q)$$

where $F_S^{WS}(q)$ is the inner service failure CDF. However, we are specifically interested in the *service's response probability* α_S^{WS} , which only refers to the inner service of a WS, and does not take into account the WS container, thus yielding

$$\gamma^{WS} = A_C^{WS} \alpha_S^{WS}. \quad (1)$$

To correctly characterize α_S^{WS} , we need to distinguish between the user's and the architect's perspectives. The user is interested in the behavior of the WS stack (see Fig. 2b), up until the BPEL-SOAP engine. We consider a *valid response* to be one that is received by the BPEL-SOAP engine, be it a service response or a BPEL-SOAP fault. On the other hand, for an architect, a response is only considered valid if it arrives from the service, be it correct or faulty. Thus, if Y is the inner service's time-to-response random variable (rv) with CDF $F_Y^{WS}(y)$, we can define the user service response probability as

$$\begin{aligned} \alpha_S^{WS^U}(y) &= Pr\{\text{The service provide a response **or** a BPEL-SOAP error occurs}\} \\ &= Pr\{\text{The service provides a response}\} + Pr\{\text{A BPEL-SOAP error occurs}\} \\ &\quad - Pr\{\text{The service provides a response **and** a BPEL-SOAP error occurs}\}. \end{aligned}$$

The inner service provides a response if it is reliable during the request processing, i.e., during the *service mission time*. Thus, assuming the software is not affected by aging, i.e. we only consider random failure causes that do not depend on the service's age, we have that $Pr\{\text{The service provides a response}\} = Pr\{T > y | Y \leq y\}$. This way, if the inner service time-to-failure and time-to-response random variables are statistically independent, we have that $\alpha_S^{WS^U}(y) = R_S^{WS}(y) + E_{BPEL-SOAP}^{WS} - R_S^{WS}(y)E_{BPEL-SOAP}^{WS}$, where $E_{BPEL-SOAP}^{WS}$ is the probability of a BPEL-SOAP error. The reader can refer to [14], [30], [31] for insights into the evaluation of $E_{BPEL-SOAP}^{WS}$.

From the architect's viewpoint, the inner service's response probability is

$$\alpha_S^{WS^A}(y) = Pr\{\text{The service provides a response}\} = R_S^{WS}(y).$$

Considering the *mean time-to-response* $\bar{y} = \int_0^\infty y T_Y^{WS}(y) dy$, we define $\alpha_S^{WS^U}$, and $\alpha_S^{WS^A}$ as $\alpha_S^{WS^U} = \alpha_S^{WS^U}(\bar{y}) = R_S^{WS}(\bar{y}) + E_{BPEL-SOAP}^{WS} - R_S^{WS}(\bar{y})E_{BPEL-SOAP}^{WS}$, and $\alpha_S^{WS^A} = \alpha_S^{WS^A}(\bar{y}) = R_S^{WS}(\bar{y})$. Thus, the WS's response probability γ^{WS} , defined by (1), is $\gamma^{WS^U} = A_C^{WS} \alpha_S^{WS^U}$ for the user's perspective, and $\gamma^{WS^A} = A_C^{WS} \alpha_S^{WS^A}$ for the architect's perspective.

Propagation matrix: The second means we use to characterize our WS system is the process' propagation matrix, which is the matrix containing the probabilities that certain inputs will be propagated to certain outputs.

The main difference between an abstract workflow and a WS is that a workflow only has two possible outputs: it can be either correct, or erroneous. A WS, on the other hand, can distinguish between correct and faulty outputs, and discriminate amongst faults by returning fault messages. These faults correspond to errors that are detected during elaboration, and managed by the process through specific handlers. There are however cases in which a process can generate an error that is not detected. Thus, following the taxonomy presented in [29], an error can also be undetected or *latent*. Some examples of undetected-latent service errors are QoS violations, errors related to functional correctness (e.g., wrong currency in payments, truncated names or strings, valid but wrong credit numbers or credentials, etc.), and errors in the WS stack (e.g., data corruption without consequences on the format, default parameters or responses

or both, etc.). Moreover, a process can also terminate before reaching the end of the workflow by invoking specific BPEL exit activities, which we characterize as *early exits*.

We specify the propagation probability matrix as

$$\mathbf{P}^{WS} = [\mathbf{P}^{WS}[i, j] = Pr\{\text{Output} = j \mid \text{Input} = i \text{ and the inner service provides a response}\}]$$

where $i, j = 0, \dots, n$ specify the input and output types ($i = 0$ identifies the correct input, $j = 0$ identifies the desired output, $0 < i, j \leq n$ identify erroneous inputs and outputs). $\mathbf{P}^{WS}[i, j]$ therefore represents the probability that a given input of type i is modified by the WS processing into an output of type j . This probability value could be statistically obtained by observing and classifying the WS output assuming the input is always of type i , for example by adopting fault propagation analysis techniques [32]. It is thus necessary to compare the actual output to the desired one, to detect the processing flaw, and to identify and classify, if possible, the output type. This action requires a clear, unambiguous input-output type classification, i.e. it should not occur that two inputs (or two outputs) of a specific WS are undistinguishable or similar.

If n_l is the number of latent errors, and n_f is the number of explicit errors or faults and early exits, then n is equal to $n_l + n_f + 1$ (1 is for the correct input-output), and matrix \mathbf{P}^{WS} can be represented as a block matrix

$$\mathbf{P}^{WS} = \left[\begin{array}{c|c} \mathbf{P}_{cl}^{WS} & \mathbf{P}_f^{WS} \\ \hline \mathbf{0}_{n_f \times n_l + 1} & \mathbf{I}_{n_f} \end{array} \right] \quad (2)$$

where \mathbf{P}_{cl}^{WS} ($n_l + 1 \times n_l + 1$) and \mathbf{P}_f^{WS} ($n_l + 1 \times n_f$) are the matrices that probabilistically represent the propagation of a correct-latent error input to the output. \mathbf{P}_{cl}^{WS} contains the correct-latent error propagation probabilities, while \mathbf{P}_f^{WS} expresses the fault or early exit ones. Assuming an incoming fault can be forwarded by the service as it is, $\mathbf{0}$ is a $(n_f \times n_l + 1)$ -dimension matrix of 0s representing the probability that a faulty input is propagated to a correct or latent error output. \mathbf{I}_{n_f} is the identity matrix of order n_f , meaning that an incoming fault is propagated as it is to the output, with probability 1.

The structure of the propagation matrix differs according to the perspective. From the end user's perspective the BPEL-SOAP fault is still considered to be a valid response. It

is a kind of error that returns the control to the users, i.e. it can be viewed in the model as a further fault. In algebraic terms, to obtain the user propagation matrix \mathbf{P}^{WS^U} , we need to add a row and a column to \mathbf{P}^{WS} of (2). This action leads us to the $n+1 \times n+1$ matrix

$$\mathbf{P}^{WS^U} = \left[\begin{array}{c|c} \alpha_S^{WS^U} \mathbf{P}_{cl}^{WS} & \alpha_S^{WS^U} \mathbf{P}_f^{WS} \mid (1 - \alpha_S^{WS^U}) \mathbf{1}_{n_l+1 \times 1} \\ \hline \mathbf{0}_{n_f+1 \times n_l+1} & \mathbf{I}_{n_f+1} \end{array} \right]$$

where $\mathbf{1}_{1 \times n_l+1}$ is a $(1 \times n_l + 1)$ -vector of 1s, and \mathbf{I}_{n_f+1} is the $n_f + 1$ -identity matrix.

As discussed in [19], the reliability of a component strictly depends on its usage. In our framework, a process or WS usage profile has to be characterized with respect to the set of failure modes. The *usage profile* or *input vector* \mathbf{In}^{WS} of a process or WS is an n -element stochastic vector. Its i th element $\mathbf{In}^{WS}[i]$ represents the probability that the input of WS carries the (correct or erred) response mode i . Because the WS manages faults, it is not possible to have faults as inputs. As a result, $\mathbf{In}^{WS}[i] = 0 \ \forall i = n_l + 1, \dots, n$ or $\mathbf{In}^{WS} = \{\overline{\mathbf{In}^{WS}}, \mathbf{0}\}$, where $\overline{\mathbf{In}^{WS}}$ is a stochastic vector of $n_l + 1$ elements, and $\mathbf{0}$ is a vector of n_f 0.

Furthermore, $\mathbf{In}^{WS^A} = \mathbf{In}^{WS}$ (, and $\mathbf{P}^{WS^A} = \mathbf{P}^{WS}$), $\mathbf{In}^{WS^U} = \{\mathbf{In}^{WS}, \mathbf{0}\}$ because, from the user's point of view, we need to include an input for the BPEL-SOAP fault.

Aggregated Parameters: Starting from γ^X and \mathbf{P}^X , we can derive the aggregated properties for a generic activity X , which could be either a WS WS or an internal activity Act , as reported in Table I.

Because $\sum_{j=0}^n \mathbf{P}^X[i, j] = 1$, from B_i^X , SE_i^X , and SF_i^X formulae, we have that $\forall i \mid 0 < i \leq n_l \ B_i^X + SE_i^X + SF_i^X = \gamma^X$.

In case of a WS WS , the above metrics are valid both from the user and the architect perspectives. Corresponding formulae can be obtained by just substituting the related parameters (γ^{WS^U} , \mathbf{P}^{WS^U} , \mathbf{In}^{WS^U} or γ^{WS^A} , \mathbf{P}^{WS^A} , \mathbf{In}^{WS^A} , respectively) into the formulae of Table I.

Assumptions: The overall approach relies on the following key assumptions.

- i) Container and inner service time-to-failure are statistically independent.
- ii) Inner service time-to-response and time-to-failure are statistically independent.
- iii) All branching activities, i.e. any activity with inner conditions (branches and

TABLE I
Aggregated parameter definitions.

Par.	Formula	Description
C_X - Correctness	$\mathbf{P}^X[0,0]\gamma^X = \mathbf{P}_{cl}^X[0,0]\gamma^X$	probability that X produces a correct output given it was invoked providing a correct input
E_i^X - i th error probability	$\mathbf{P}^X[0,i]\gamma^X = \mathbf{P}_{cl}^X[0,i]\gamma^X$	probability that a correct invocation/execution of X returns with an error mode i with $0 < i \leq n_l$
G_j^X - j th fault probability	$\mathbf{P}^X[0, n_l + j]\gamma^X = \mathbf{P}_f^X[0,j]\gamma^X$	the probability that a correct invocation triggers the j th fault on X with $0 < j \leq n_f$
B_i^X - i th error robustness	$\mathbf{P}^X[i,0]\gamma^X = \mathbf{P}_{cl}^X[i,0]\gamma^X$	the probability that X , invoked with an error mode i , $0 < i \leq n_l$, masks the error and returns a correct output
SE_i^X - i th error susceptibility	$\gamma^X \cdot \sum_{j=1}^{n_l} \mathbf{P}^X[i,j] = \gamma^X \sum_{j=1}^{n_l} \mathbf{P}_{cl}^X[i,j]$	probability that X , invoked with an error mode i , $0 < i \leq n_l$, produces an erroneous output
SF_i^X - i th fault susceptibility	$\gamma^X \cdot \sum_{j=1}^{n_f} \mathbf{P}^X[i, n_l + j] = \gamma^X \sum_{j=1}^{n_f} \mathbf{P}_f^X[i,j]$	probability that X , invoked with an error mode i , $0 < i \leq n_l$, produces a fault output
L_i^X - i th proclivity	$\gamma^X \sum_{h=0}^{n_e} \mathbf{In}^X[h] \mathbf{P}^X[h,i]$	probability that X produces the i th error mode ($0 < i \leq n_l$) or fault output ($n_l < i \leq n_f$) given \mathbf{In}^X . In the former case ($0 < i \leq n_l$), we refer to <i>error proclivity</i> , while in the latter case ($n_l < i \leq n_f$) we refer to <i>fault proclivity</i> .

loops), can be probabilistically represented.

iv) Input and output types need to be distinguishable. Given an input or output activity of type i ($\mathbf{In}[i]$ or $\mathbf{O}[i]$), $\forall j \in [1, n] | i \neq j$ it is true that $\mathbf{In}[i] \neq \mathbf{In}[j]$ or $\mathbf{O}[i] \neq \mathbf{O}[j]$. The absence of this assumption would make it impossible to define the propagation matrix.

v) To deal with loops, we assume that the input of the loop body is the same for all the iteration steps (\mathbf{In}^L). The assumption is that, at each iteration, the same input will be processed, and not the output of the previous iteration. This assumption is quite realistic in composite processes, because loops often process the same input until the required results are obtained (e.g. a reservation or a payment is iterated until success), or the input is provided to the loop as aggregated data and processed one block at a time.

vi) Due to the fact that the model should be finite, we fix the maximum number of events that can be handled at the same time by an *event handler*. Thus, at most $n_e > 0$ events can be handled together by the system. Even if this is an approximation, it is important to remark that n_e can be fixed arbitrarily large to best tune the tolerance, without any significant impact on the algorithm complexity.

5 FLATTENING BPEL PROCESSES

In this section, we detail the second step of our evaluation technique, and explain how we obtain a flat workflow starting from the original BPEL composition. This transformation is necessary for two reasons. The first reason is that it allows us to provide unambiguous semantics for BPEL activities, and to clarify the complex control- and data-flows that emerge when various compensation-, fault-, and event-handlers are attached to nested scopes. The second reason is that the flat workflow is self-contained; it includes all the information required to achieve the analysis, can be used with no further elaboration, and is suitable for an automatic tool.

The following discussion is exemplified using a travel management application as a running example.

5.1 A BPEL Primer

BPEL 2.0's main constructs (called activities) can be classified as *basic* or *structured*. A compact explanation is reported in Table II. It also highlights the graphical notation adopted in the rest of the article to represent BPEL processes. *Basic activities* implement elementary steps of a process workflow. More specifically, `<invoke>`, `<receive>`, and `<reply>` activities manage service interactions, while the `<assign>`, `<validate>`, `<wait>`, `<empty>`, and `<exit>` activities are self-explanatory. *Structured activities* represent control-flow logic structures. `<sequence>`, `<if-elseif-else>`, `<while>`, `<repeatUntil>`, and `<forEach>` are self-explanatory. The `<pick>` activity forces the process to wait for the delivery of a message (`<onMessage>` construct) or a timeout expiration (`<onAlarm>` construct) to perform the activities associated with the corresponding branch. Finally, the `<flow>` construct introduces activities that have to be performed concurrently. The designer can use a `<scope>` construct to define nested activities.

BPEL also provides designers with mechanisms, called handlers, for capturing and dealing with special kinds of events, which we can classify as faults, needs for compensation, or concurrent events. Fault handlers can be used to capture and deal with runtime faults. They are defined through `<catch>` and `<catchAll>` constructs that are associated to a `<scope>`. When a fault is caught, the `<scope>` is terminated, and the corresponding handler is launched. If a fault is not caught by an appropriate handler,

TABLE II
BPEL 2.0 basic (a) and structured (b) activities.

BPEL activity	Symbol	Description
<code><assign></code>		assign values to variables
<code><validate></code>		validate the state of variables
<code><wait></code>		wait for the specified amount of time
<code><invoke></code>		synchronously (up) or asynchronously (down) call a partner WS
<code><receive></code>		wait for a message
<code><reply></code>		send a response message
<code><throw></code>		raise a fault
<code><rethrow></code>		re-throw a fault to the upper level scope
<code><empty></code>		do nothing
<code><exit></code>		immediately terminate the business process

(a)

BPEL activity	Symbol	Description
<code><sequence></code>		sequence of activities
<code><if></code>		select exactly one activity from a set of choices
<code><while></code>		loop structure
<code><repeat-Until></code>		loop structure
<code><pick></code>		wait for one of several messages to arrive or for a timeout to occur
<code><flow></code>		concurrent execution of activities
<code><forEach></code>		loop structure
<code><scope></code>		define an execution scope
<code><compensateScope></code>		start compensation on an inner scope
<code><compensate></code>		start compensation on all inner scopes

(b)

it is re-thrown to an upper level. If the handler completes successfully, the control flow returns to the activity that immediately follows the `<scope>`. Faults can also be thrown using the `<throw>` construct. Compensation handlers are used to define compensation logic, and they can only be activated for scopes that have completed successfully. They are explicitly initiated using the `<compensate>` or `<compensateScope>` activities.

Finally, event handlers consist of business logic that can be activated concurrently to the process' main logic. The activities within a handler are triggered either by the arrival of an inbound event message, or by a timed alarm.

Fig. 3 shows the BPEL process of a travel management service example. Once a request

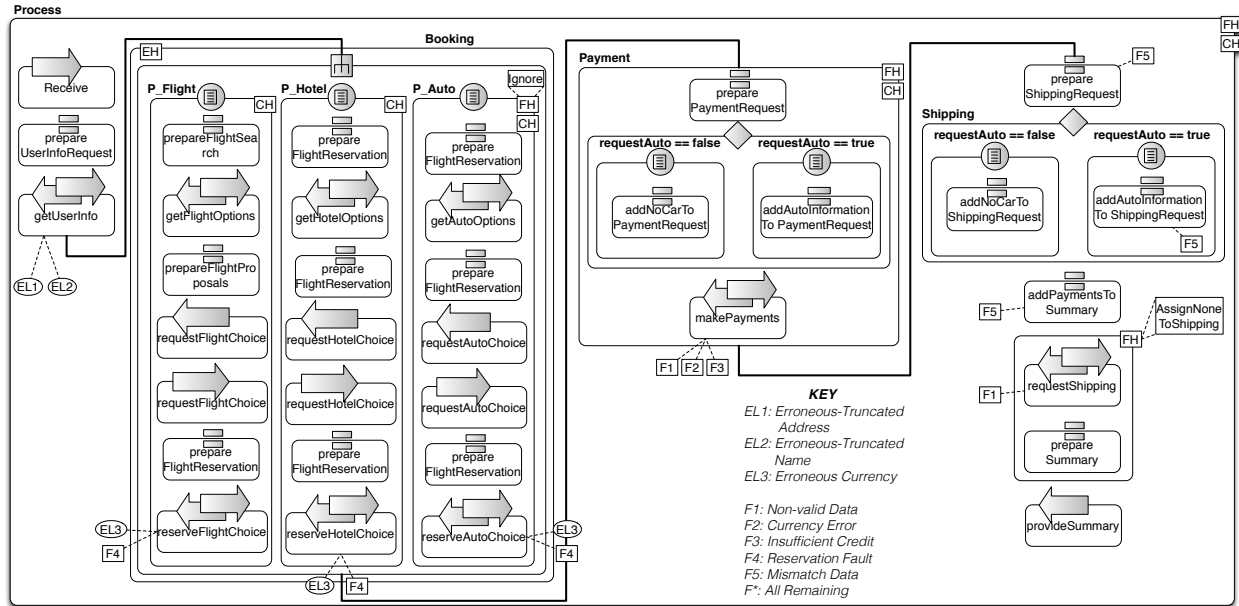


Fig. 3. The running example BPEL process model.

is received from the user and assigned to a local variable, a query on the user DB is performed to retrieve user data. In the case of faults, the process immediately exits and returns a fault code. Otherwise, the user request is processed by invoking a three-way *<flow>*, in which the process concurrently elaborates the flight, hotel, and (optional) car booking. The booking status can be checked by invoking the event handler specified by the *Booking <scope>*. Each service involved in the processing can fail, thus each branch is isolated from the others through a specific *<scope>*. The fault handlers associated with the first two *<flow>* branches compensate the activities of the *<scope>* that are already complete, and forward the fault to the upper layer. The third branch defines a fault handler that masks faults so that they are not forwarded to the process, thus totally isolating the branch from the process.

When the three subprocesses finish, the *makePayments* service is invoked. If there are problems in the *Payment*, the fault handler associated with its *<scope>* tries to recover the fault by nesting three levels of fault handlers, and by canceling the car reservation in the case of wrong currency or insufficient credit. Finally, in the case of successful payment, the shipping service is invoked (*requestShipping*) within its

own *<scope>*, to isolate it and to mask faults that should not reach the process. Details on the fault, compensation, and event handling can be found in [33].

5.2 BPEL Process to Flat Workflow

In this section, we will clarify how we flatten a BPEL process to a workflow that only uses the three basic patterns of structured programming: (*sequence*, *branch* and *loop*), and concurrency (*fork-join*). To explain the BPEL to workflow mapping, we can identify two steps. First we detail how to map single BPEL activities into corresponding workflows. Second, we describe how the whole workflow can be obtained by composing the single activity workflows. The mapping rules and the algorithms that we obtain are then applied to the running example.

5.2.1 Mapping BPEL Activities

BPEL flattening can be performed by extending and hierarchically applying the flattening rules that deal with single BPEL activities.

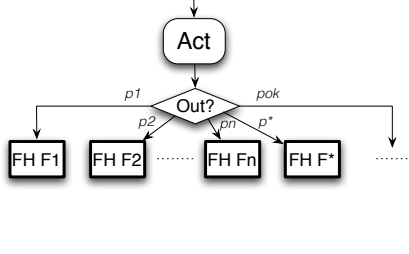
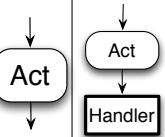

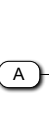
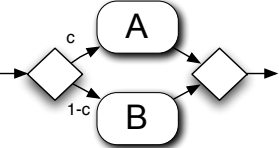
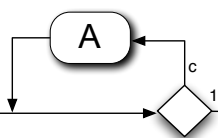
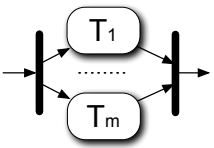
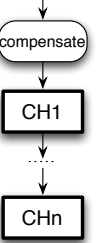
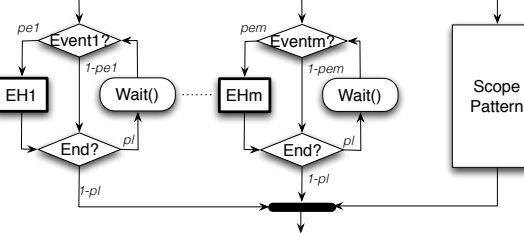
As shown in Table II, BPEL activities are split into two main classes: basic, and structured. Basic activities such as *<wait>*, *<invoke>*, *<receive>*, *<reply>*, and *<empty>* do not modify the process workflow so they can be considered as internal activities that do not affect a process' reliability. On the other hand, *<assign>* and *<validate>* can trigger internal errors such as datatype mismatches, and therefore they need to be considered as faulty activities.

Similarly, *<throw>* and *<rethrow>* activities are used in fault handling to manage faults, and thus they can change the process workflow and its reliability; the same is true for *<compensate>* and *<compensateScope>* activities.

The *<exit>* activity forces the process termination, thus impacting on the process workflow. This effect implies that a BPEL process may have different possible outputs, and therefore it is necessary to evaluate the probability of each of them.

Structured activities mainly implement control flow operations. Most of them (*<sequence>*, *<if>*, *<while>*, *<repeatUntil>*, *<forEach>*, *<pick>*, and *<flow>*) do not specifically involve any flattening operation, while further investigation is required for the *<compensate>* and *<scope>* BPEL structured activities.

TABLE III
BPEL 2.0 activities, and corresponding flat workflows.

<assign>, <validate>, <invoke>, <receive>	<wait>, <empty>	<throw>, <rethrow>, <compensateScope>	<reply>, <exit>	<sequence>	<if>, <pick>
					
<while>, <repeatUntil>, <forEach>	<flow>	<compensate>	<scope>		
					

<Scope> is a powerful BPEL construct, through which one can specify a context influencing the execution behavior of its enclosed activities. Such contexts or <scope>s can be nested hierarchically, while the root context is provided by the <process> itself. In particular, a behavioral context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler.

Table III shows the flattened workflows that correspond to the BPEL 2.0 activities we have discussed. Basic activities are represented by rounded rectangles, while thick squared boxes represent subprocesses that are composed of the other activities, which in turn can also be subprocesses. As for basic activities, we mainly characterize non-

faulty activities ($\langle wait \rangle$, $\langle empty \rangle$) through a simple one input to one output activity. Because they do not affect the BPEL workflow's reliability, they are usually not reported in the process workflow. Faulty activities ($\langle assign \rangle$, $\langle validate \rangle$, $\langle invoke \rangle$, $\langle receive \rangle$), on the other hand, can generate faults due to internal errors ($\langle assign \rangle$, $\langle validate \rangle$) or to external faults triggered by WS invocations. What we do in the flattening process is to merge the WS behavior with the process' interaction, i.e, we represent the WS as embedded inside an $\langle invoke \rangle$ (in case of synchronous WS interactions) or a $\langle receive \rangle$ (in case of asynchronous WS interactions). Thus, we can argue that $\langle invoke \rangle$ and $\langle receive \rangle$ are intrinsically not faulty activities, but that they reflect the associated WS behavior.

The probabilities p_1, p_2, \dots, p_n , and p_{ok} shown in the faulty activity workflow are related to the occurrence of a specific fault, or to the probability of a correct or latent error output. To obtain these probabilities, we need to know the user profile input probability vector, as we will see in Section 6.

$\langle Throw \rangle$, $\langle rethrow \rangle$, and $\langle compensate \rangle$ activities are considered as explicit invocations of specific FCT handlers, and therefore are represented accordingly. The $\langle reply \rangle$ and $\langle exit \rangle$ activities are instead mapped as final activities that close the process. Structured activities are considered as specific patterns to be applied to basic activities, and the corresponding flattened out workflows are reported in Table III. More specifically, the $\langle compensate \rangle$ activity is mapped into a sequence of compensation handlers that compensate the behavior of all the successfully completed scopes that are immediately enclosed inside the scope associated with the FCT-handler. Finally, a $\langle scope \rangle$ is mapped as a parallel activity with $m+1$ branches: the first m branches describe the event handling and management of the corresponding events e_1, \dots, e_m , while the last branch is related to the $\langle scope \rangle$ subprocess mapping.

5.2.2 Obtaining the Workflow

Once the mapping of the BPEL elements has been completed, they are combined into the flat workflow. The algorithms shown in Fig. 4 describe the actions that need to be taken to obtain this workflow. Let us start by explaining the algorithm that deals with the whole process, which is shown in Fig. 4a.

Each activity Act of a process or subprocess is elaborated through the $Map(Act, ScopeHP)$

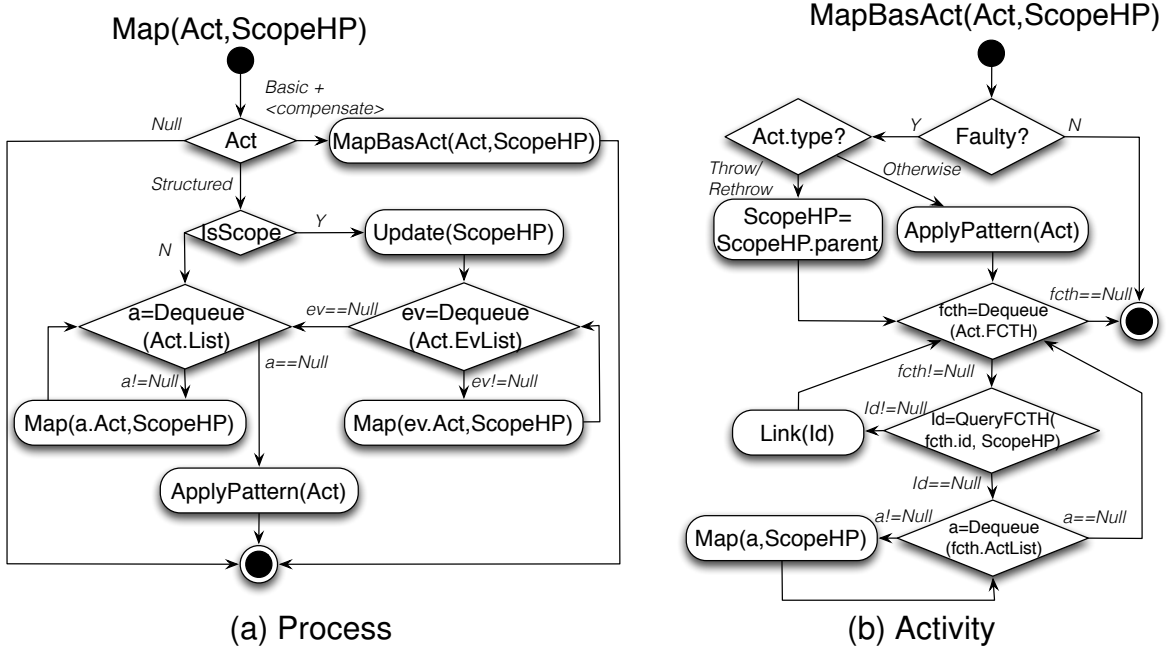


Fig. 4. The mapping algorithms.

function, according to whether it is basic or structured. *ScopeHP* is the scope hierarchy path, i.e., the path in the scope hierarchy of the considered activity *Act*, which allows us to determine the FCT handler workflows. In the case of basic or *<compensate>* activities, the *MapBasAct* function of Fig. 4b is performed. If the structured *Act* is a *<scope>*, the *ScopeHP* has to be updated with the addition of the new scope; if the scope has handlers they too need to be recursively mapped using the *Map* function. If the structured *Act* is not a *<scope>*, or if the *<scope>* event handler mapping loop is completed, a loop is performed to map all the activities that are nested in the *Act*. Finally, the corresponding workflow pattern, among those reported in Table III, is applied to the activities enclosed within *Act* (as well as to the *<scope>* handler). This step completes the process workflow.

The *MapBasAct(Act, ScopeHP)* of Fig. 4b maps basic or *<compensate>* activities into the corresponding workflow. It starts by checking whether the considered activity *Act* is faulty or not. Non-faulty activities that do not modify the input, i.e., with identical propagation matrices (internal activities such as *<assign>* and *<validate>*), except *<throw>*, *<rethrow>*, *<compensate>*, and *<compensatescope>*, are immediately skipped. The others are forwarded and processed as faulty activities.

Thus, the workflow pattern associated with *Act* in Table III is applied, and the *Act* handlers are mapped into the corresponding workflows. Because faults occurring to activities within the same $\langle scope \rangle$, or within the same *ScopeHP*, invoke the same fault handlers, the fault handlers are mapped into the corresponding workflow the first time they are invoked. The invocations on these fault handlers return the workflow identifiers as pointers to the corresponding workloads. In case a fault handler in the specific *ScopeHP* has not yet been invoked, the mapping is performed by considering all the activities enclosed in the fault handler. This is why the two functions implement a mutual recursion.

We have applied these algorithms to the running example shown in Fig. 3. Internal errors and faults are usually much less frequent than WS errors and faults; therefore, we assumed internal activities were error and fault-free. Thus, $\langle assign \rangle$ and $\langle validate \rangle$ activities are not considered as faulty, and are not taken into account in the example. This way, fault *F5*, which is related to a data mismatch and affects $\langle assign \rangle$ activities, can be neglected, as it has a very low probability of occurring. The workflow only includes the synchronous invokes, and the corresponding WS faults. By flattening the running example's BPEL, we obtain the workflow shown in Fig. 5. Because the main process fault handlers ignore inner faults, they are represented as exit activities that bypass the compensation handlers. Furthermore, event handling is considered to be a request to check the *Booking* status at a certain time. Handling the faults triggered by the *makePayments* service is more complex, because they also have nested scopes and fault handlers. Their workflows are highlighted in Fig. 5. Specifically, the *makePayments* invocation can throw 4 different faults: *F1*, *F2*, *F3*, and the generic *F**. The most complex one to manage is *F1*, as it may throw *F2*, *F3*, *F** (handled by the *makePayments* calling $\langle scope \rangle$), and *F1* (handled by the process $\langle scope \rangle$) faults. Also, the *F3* handler is nested because it may cause a generic fault *F** to be thrown to the upper *makePayments* $\langle scope \rangle$. Finally, *F2* and *F** have simple fault forwarding handlers that notify the fault to the end user. This example allows us to show how we transform a complex, hierarchical, nested BPEL process into a flat workflow that is ready to be evaluated.

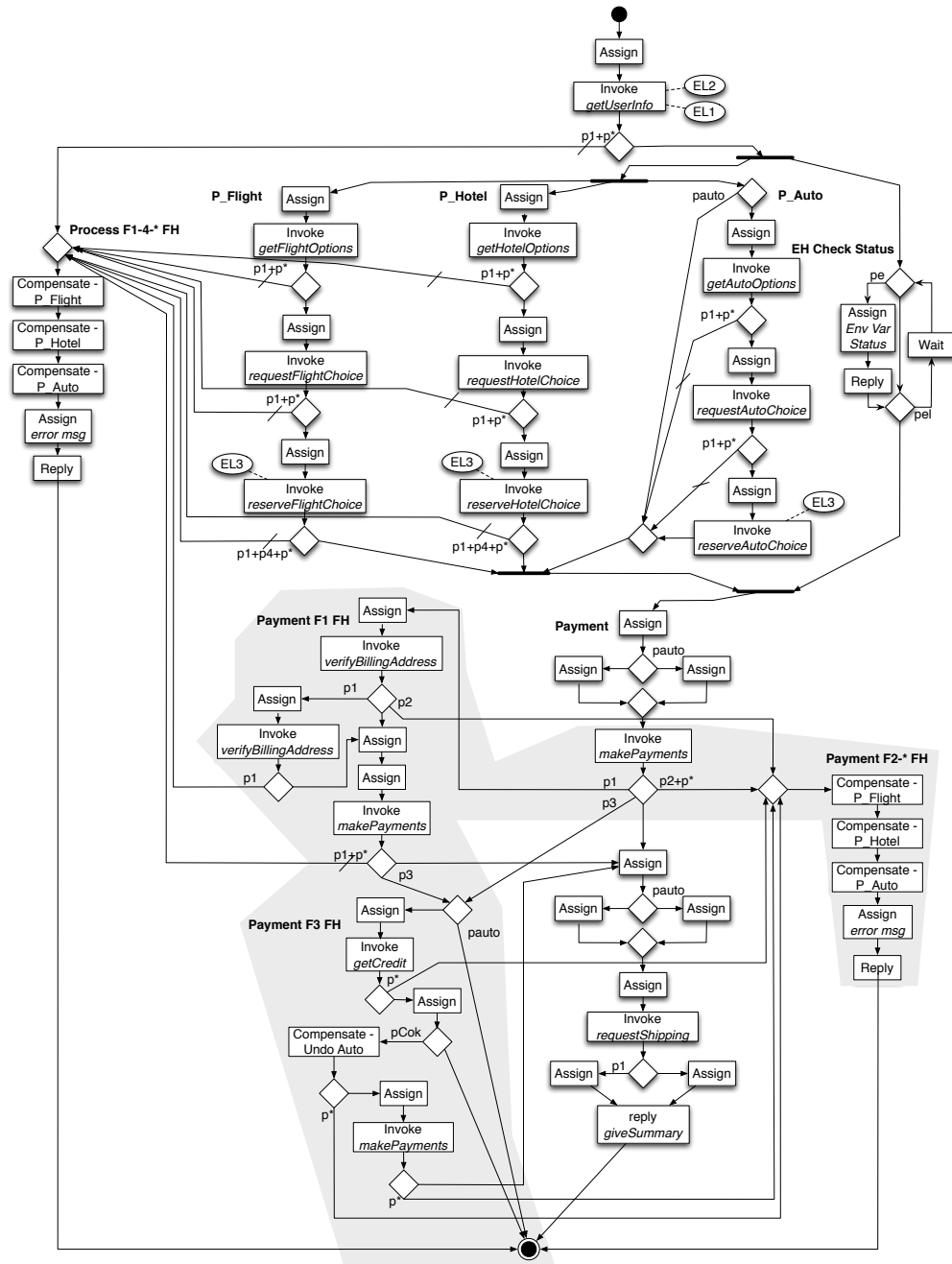


Fig. 5. Running Example process workflow.

6 SYNTHESIS AND ANALYSIS

Once we have obtained a flat workflow, we are ready to analyze it, taking into consideration any faults and fault handlers, from both the end user's and the architect's perspectives.

6.1 Fault Handling

In BPEL processes, faults can be managed by ad-hoc handlers, which can therefore be used to stop fault propagation. Fault handlers can be broadly categorized into *forwarders*, and *processors*. The former simply manage the termination of the process, and send the user a message containing the details of the fault. The latter implement some specific recovery actions that can either be successful or require further fault handling. A fault handler can be viewed as a subprocess that elaborates on the main workflow's information to recover from a fault. In terms of propagation effects, this approach results in a feedback of the process input into the main workflow, after the fault has been handled.

Thus, assuming $\mathbf{In} \in \mathbb{R}^n$ is the input probability vector of the process, i.e. the usage profile of Section 4, and that $\mathbf{O} \in \mathbb{R}^n$ is its output probability vector, we can algebraically express the process propagation as an input-output relationship by the function $F(\cdot)$

$$\mathbf{O} = F(\mathbf{In}, \mathbf{P}^1, \dots, \mathbf{P}^m) \quad (3)$$

where $\mathbf{P}^1, \dots, \mathbf{P}^m$ are the propagation matrices of the $m > 0$ process activities, identified by $i = 1, \dots, m$. $F : \mathbb{R}^n \times \underbrace{\mathbb{R}^{n,n} \times \dots \times \mathbb{R}^{n,n}}_m \rightarrow \mathbb{R}^n$ returns an n -probability vector to obtain output $j \leq n$ from input $i \leq n$ given that a response is provided, as stated in Section 4.

Because fault forwarders simply forward faults, they avoid input feedback. So, if all the fault handlers in the process are fault forwarders, the function of (3) becomes

$$\mathbf{O} = \mathbf{In} \cdot F_{eq}(\mathbf{P}^1, \dots, \mathbf{P}^m) \quad (4)$$

where $F_{eq} : \underbrace{\mathbb{R}^{n,n} \times \dots \times \mathbb{R}^{n,n}}_m \rightarrow \mathbb{R}^{n,n}$ returns the system *equivalent matrix* $\mathbf{P}_{eq} = F_{eq}(\mathbf{P}^1, \dots, \mathbf{P}^m)$.

The output is therefore linear with respect to the input. Hence, it could be obtained through simple algebraic operations on the workflow, as shown in Section 6.2.

On the other hand, if there are one or more fault processors in the process, we have input feedback, meaning that the relationship between input and output is not linear. In other terms, in the case of fault forwarding, correct response or latent error flows and faults are separated. Indeed, they are forwarded by the main workflow to the output without being mixed. In the case of fault processing, some faults can be processed and

recovered, so becoming correct responses or latent errors. This policy leads to fault and correct or latent error flows being mixed in the propagation.

To adequately investigate and characterize the problem, let us start by considering a generic faulty activity $X = 1, \dots, m-1$, and its fault management. As stated above, we can express the activity output probabilities as functions of the input $\mathbf{O}^X = F^X(\mathbf{In}^X, \mathbf{P}^X) = \{p_{O_0^X}, p_{O_1^X}, \dots, p_{O_{n_l}^X}, p_{O_{n_l+1}^X}, \dots, p_{O_{n_l+n_f+1}^X}\}$. Table III shows that this characterization specifically regards faulty activities, both internal (*<assign>*, *<validate>*), and external (synchronous *<invoke>*, *<receive>*), where probabilities $p_{F_1}, p_{F_2}, \dots, p_{F_{n_f}}$ correspond to $p_{O_{n_l+1}^X}, p_{O_{n_l+2}^X}, \dots, p_{O_{n_l+n_f+1}^X}$, respectively, and $p_{ok} = \sum_{j=0}^{n_l} p_{O_j^X}$. To obtain the $(X+1)$ th activity input (\mathbf{In}^{X+1}), following the X th one, we have to know the X normalized output $\overline{\mathbf{O}^X}$.

$$\mathbf{In}^{X+1} = \overline{\mathbf{O}^X} = \{p_{O_0^X}/p_{ok}^X, p_{O_1^X}/p_{ok}^X, \dots, p_{O_{n_l}^X}/p_{ok}^X, 0, \dots, 0\} = \left\{ \frac{\mathbf{In}^X \cdot \mathbf{P}_{cl}^X}{p_{ok}^X}, 0 \right\} = \{\overline{\mathbf{In}^{X+1}}, 0\} \quad (5)$$

where $\overline{\mathbf{In}^{X+1}} = \frac{\mathbf{In}^X \cdot \mathbf{P}_{cl}^X}{p_{ok}^X} \in \mathbb{R}^{n_l+1}$, and therefore $\sum_{j=0}^{n_l} \overline{\mathbf{In}^{X+1}}[j] = 1$.

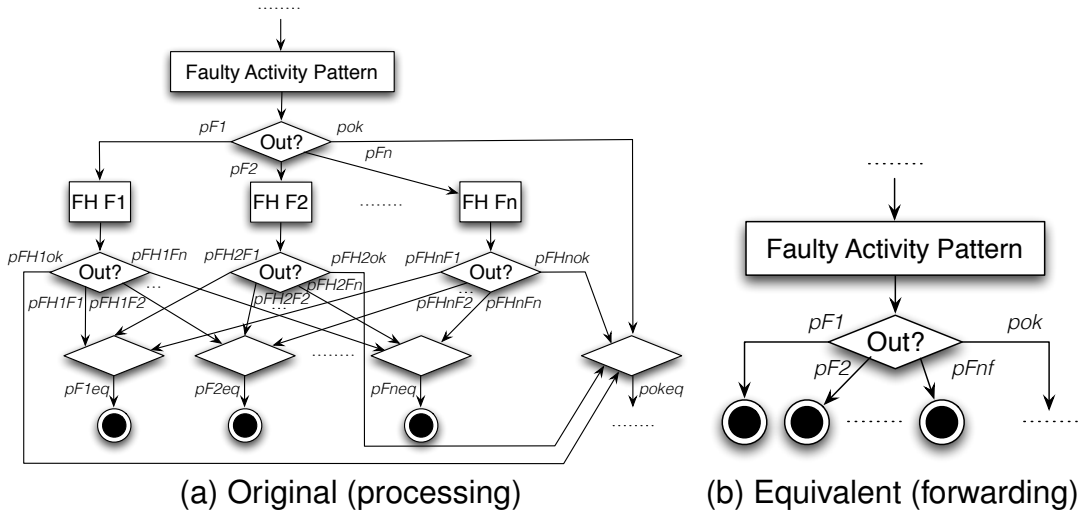


Fig. 6. Fault handler workflow pattern.

Fig. 6a shows a generic fault processing workflow, in which each fault is managed by an explicit fault handler. Notice that inner faults and their handlers are collapsed into the main fault handler. Nevertheless, the proposed solution can be inductively applied to nested fault handlers, as we shall see. Our aim here is to obtain a simpler model, like the one shown in Fig. 6b, which is *equivalent* to the one shown in Fig. 6a. To this end,

we have to identify the $p_{F1eq}, \dots, p_{Fn_feq}$, and p_{okeq} probabilities of the equivalent model, given the original model's probabilities of faulty activity output (p_{Fi} for the generic i th output), and of all output for each of the fault handlers, i.e. p_{FHjFk} for the generic k th output of the j th fault handler with $i, j, k = 0, \dots, n_f$ (conventionally, $p_{F0} = p_{ok}$, $p_{FHjF0} = p_{FHjok}$, and $p_{FH0Fk} = p_{ok} \mathbf{In} \cdot \mathbf{P} \cdot \mathbf{I}_n[k]$). In other words, considering a generic faulty activity X with propagation matrix \mathbf{P}^X , we want to obtain the equivalent model characterized by \mathbf{P}_{eq}^X , so that the output $\mathbf{O}^X = F(\mathbf{In}^X, \mathbf{P}^X)$ of (3) is expressed as

$$\mathbf{O}^X = \mathbf{In}^X \cdot \mathbf{P}_{eq}^X(\mathbf{In}^X),$$

just like in the case of fault forwarding.

There is a strong difference between this relationship and the one seen in (4), i.e. $\mathbf{P}_{eq}^X(\mathbf{In}^X) = F_{eq}^X(\mathbf{P}^{FH0}, \dots, \mathbf{P}^{FHn_f}, \mathbf{In}^X)$, where $\mathbf{P}^{FH1}, \dots, \mathbf{P}^{FHn_f}$ are associated to the fault handlers of faults $1, \dots, n_f$, and $\mathbf{P}^{FH0} = \mathbf{I}_n$ is the matrix that describes a fictitious handler that manages a correct response, by just forwarding it. This way the aggregation-reduction rules are still valid, and can be applied.

To obtain the simplified activity model of Fig. 6b, we have to express its probabilities $p_{F1eq}, \dots, p_{Fn_feq}, p_{okeq}$ in terms of the original model probabilities and matrices. In the fault processing we assume that the output of the faulty activity, before handling $\mathbf{O}^{X'} = \mathbf{In}^X \cdot \mathbf{P}^X$, is forwarded to the fault handlers so that they can perform recovery, thus propagating and reprocessing the normalized output $\overline{\mathbf{O}^{X'}}$, as in the normal workflow. It follows that, given $\mathbf{O}^{FHj} = \overline{\mathbf{O}^{X'}} \cdot \mathbf{P}^{FHj} = \{\overline{\mathbf{In}^X} \cdot \mathbf{P}_{cl}^X \cdot \mathbf{P}_{cl}^{FHj}, \overline{\mathbf{In}^X} \cdot \mathbf{P}_{cl}^X \cdot \mathbf{P}_f^{FHj}\}$, the output of the whole faulty activity, including the fault processing, is $\mathbf{O}^X = \mathbf{In}^X \cdot \mathbf{P}_{eq}^X = \sum_{i=0}^{n_f} p_{Fi} \mathbf{In}^X \cdot \mathbf{P}^X \cdot \mathbf{P}^{FHi} = p_{ok} \mathbf{In}^X \cdot \mathbf{P}^X \cdot \mathbf{I}_n + p_{F1} \mathbf{In}^X \cdot \mathbf{P}^X \cdot \mathbf{P}^{FH1} + \dots + p_{Fn_f} \mathbf{In}^X \cdot \mathbf{P}^X \cdot \mathbf{P}^{FHn_f}$.

The first term ($p_{ok} \mathbf{In}^X \cdot \mathbf{P}^X \cdot \mathbf{I}_n$) represents the WS output ($\mathbf{In}^X \cdot \mathbf{P}^X$), which is forwarded to the workflow in case of correct response ($p_{F0} = p_{ok}$) through a fictitious fault handler ($\mathbf{P}_{FH0} = \mathbf{I}_n$). This way the equivalent propagation matrix \mathbf{P}_{eq} can be expressed as

$$\mathbf{P}_{eq}^X(\mathbf{In}^X) = \mathbf{P}^X \cdot \left(p_{ok}^X \mathbf{I}_n + p_{F1}^X \mathbf{P}^{FH1} + \dots + p_{Fn_f}^X \mathbf{P}^{FHn_f} \right) \quad (6)$$

where $p_{ok}^X = \sum_{i=0}^{n_l} \mathbf{In}^X \cdot \mathbf{P}^X[i]$, $p_{Fj}^X = \mathbf{In}^X \cdot \mathbf{P}^X[j]$, and thus its dependence on \mathbf{In}^X .

6.2 Aggregation/Reduction Rules

Once the BPEL process activities have been mapped to equivalent activities, we can evaluate process parameters ($\langle \gamma^P(\mathbf{In}), \mathbf{P}^P(\mathbf{In}) \rangle$) by applying the reduction algorithm proposed in [25], which we have adapted for SOA and WS compositions, and specifically for dealing with FCT and event handling, taking into account the fault processing input feedback and its propagation on the main process workflow. In the following, we provide reduction rules for all the BPEL structured activity workflow patterns, considering just 2 sub-activities (A , and B) described, as specified in Section 4, by the 2-tuples $\langle \gamma^A(\mathbf{In}), \mathbf{P}^A(\mathbf{In}^A) \rangle$, and $\langle \gamma^B(\mathbf{In}), \mathbf{P}^B(\mathbf{In}^B) \rangle$, respectively. The formulae we obtain can be easily generalized to n -sub-activity patterns by iteratively applying the rules to reduced sub-activity couples, until we obtain a single equivalent component.

Both the user and the architect perspectives need to be considered to obtain the $\mathbf{P}^{P^U} \in \mathbb{R}^{n+1 \times n+1}$ and $\mathbf{P}^{P^A} \in \mathbb{R}^{n \times n}$ process propagation matrices based on those representing the activities. Furthermore, from the architect's perspective, the response probability γ^{P^A} also depends on the process workflow, and requires further investigation. Indeed, the probability of a specific workflow path has to take into account propagation effects, and can therefore be expressed as a function of the WS fault probabilities, as discussed in Section 6.1. The workflow equivalent is similar to the one shown in Fig. 7, where p_{next} is the probability to continue the workflow, while p_{out} is the probability to exit due to a fault, where $p_{next} + p_{out} = 1$, and therefore $p_{out} = 1 - p_{next}$. Moreover, if γ^{P^A} represents the response probability of the equivalent system, it has to be expressed in terms of the related parameters (γ^{A^A} , and γ^{B^A} in the case of two activities A , and B).

To this end, considering a generic process structured activity X , it is possible to express γ^{X^A} in algebraic terms by manipulating the propagation matrix $\mathbf{P}^{X^A}(\mathbf{In}^X)$

$$\mathbf{P}^{X^{A\gamma}}(\mathbf{In}^X) = \left[\begin{array}{c|c} \gamma^{X^A} \mathbf{P}_{cl}^{X^A}(\mathbf{In}^X) & \gamma^{X^A} \mathbf{P}_f^{X^A}(\mathbf{In}^X) \\ \hline \mathbf{0}_{n_f \times n_l + 1} & \mathbf{I}_{n_f} \end{array} \right]$$

thus obtaining the *architect γ -propagation matrix* $\mathbf{P}^{X^{A\gamma}}(\mathbf{In}^X)$. Note that $\mathbf{P}^{X^{A\gamma}}(\mathbf{In}^X)$ is a non-stochastic matrix because the sum by row elements of rows $1, \dots, n_l$ is equal to γ^{X^A} . Hence, to obtain parameter γ^{P^A} for the whole process, given the process input \mathbf{In}^P , we

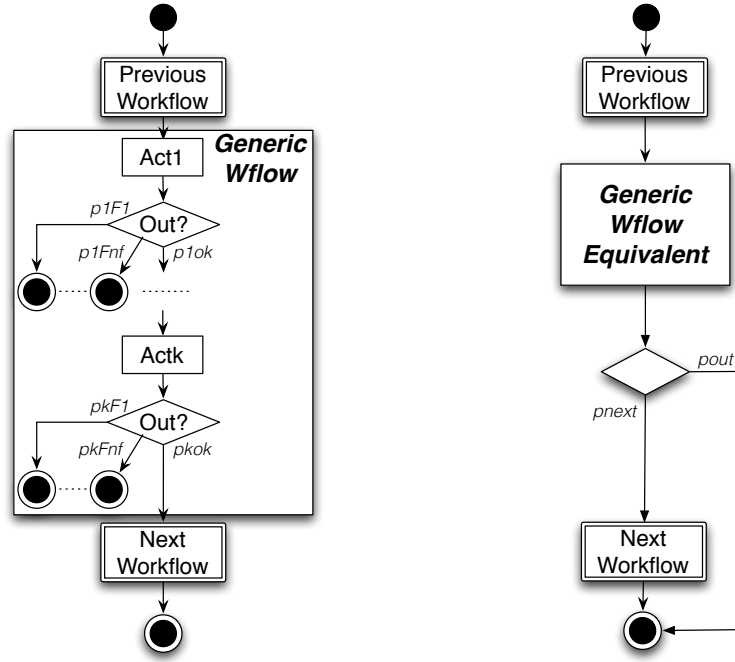


Fig. 7. A generic workflow equivalent.

have to evaluate its γ -propagation matrix $\mathbf{P}^{PA\gamma}(\mathbf{In}^P)$, and output vector so that

$$\gamma^{PA} = \sum_{i=0}^n \mathbf{In}^P \cdot \mathbf{P}^{PA\gamma}(\mathbf{In}^P). \quad (7)$$

Also, this parameter, from the architect's perspective, depends on \mathbf{In}^P , i.e. $\gamma^{PA}(\mathbf{In}^P)$.

TABLE IV
Aggregation rules.

Pattern		\mathbf{P}^P	$\mathbf{P}^{PA\gamma}$	p_{next}
Seq		$\mathbf{P}^A \cdot \mathbf{P}^B$	$\mathbf{P}^{A^{A\gamma}} \cdot \mathbf{P}^{B^{A\gamma}}$	$p_{next}^S = p_{ok}^A p_{ok}^B$
If	No exit branch	$p_c \mathbf{P}^A + (1 - p_c) \mathbf{P}^B$	$(1 - p_l)(\mathbf{I}_n - p_l \mathbf{P}^{A^{A\gamma}}(\mathbf{In}^L))^{-1}$	$p_{next}^C = p_c p_{ok}^A + (1 - p_c) p_{ok}^B$
	Exit branch	$\mathbf{P}^A(\mathbf{In}^C) \cdot \mathbf{P}^{B_{eq}}$		
Loop	While	$(1 - p_l)(\mathbf{I}_n - p_l \mathbf{P}^A)^{-1}$	$(1 - p_l)(\mathbf{I}_n - p_l \mathbf{P}^{A^{A\gamma}})^{-1}$	$p_{next}^L = \frac{p_{ok}^A}{1 - p_l p_{ok}^A}$
	RU	$(1 - p_l) \mathbf{P}^A \cdot (\mathbf{I}_n - p_l \mathbf{P}^A)^{-1}$	$(1 - p_l) \mathbf{P}^{A^{A\gamma}} \cdot (\mathbf{I}_n - p_l \mathbf{P}^{A^{A\gamma}})^{-1}$	
Concurrent		$\mathbf{P}^F = \frac{1}{2}(\mathbf{P}^A(\mathbf{In}^F) + \mathbf{P}^B(\mathbf{In}^F))$	$\frac{1}{2}(\mathbf{P}^{A^{A\gamma}} \cdot \mathbf{P}^{B^{A\gamma}}) + \frac{1}{2}(\mathbf{P}^{B^{A\gamma}} \cdot \mathbf{P}^{A^{A\gamma}})$	$p_{next}^F = p_{ok}^A p_{ok}^B$

Table IV shows the aggregation-reduction rules for the workflow patterns identified in Section 5. These have been obtained by algebraic manipulations similar to those of [4], [25]. Details can be found in [33].

6.3 Solution Algorithm

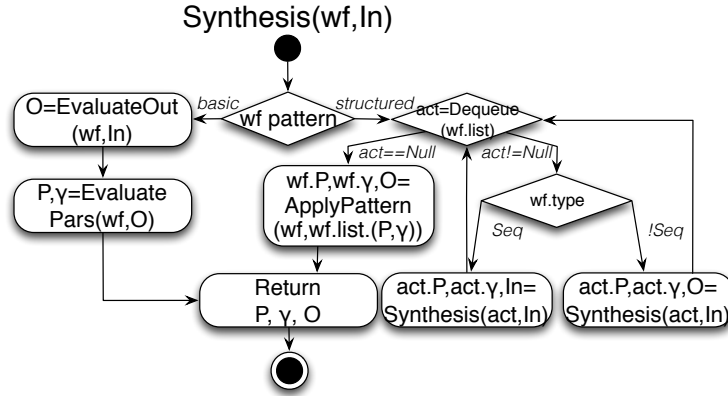


Fig. 8. BPEL parameter evaluation algorithm.

The main synthesis algorithm for evaluating a BPEL process' response probability and propagation effects is shown in Fig. 8. It receives the flat workflow wf , and the usage profile In as inputs; and returns γ , P representing the process from the required perspective, and the corresponding output O . As discussed in Section 5, wf has to include all the BPEL process' activities and fault handlers. With regards to fault handlers, they are considered by the algorithm as sequences containing the faulty activity and the fault handler activities, as depicted in Fig. 6.

The algorithm starts by processing the input workflow, to evaluate if it is composed of just a single basic activity, or if it is structured. In the latter case, we need to consider the nested activities; the function is therefore recursively invoked on all of them. In particular, if we have sequence patterns, the output of an activity becomes the input to the next activity; otherwise, if we have a loop, fork-join or flow construct, the same input is used for all the nested workflows and activities.

When all the nested workflows have been analyzed, `ApplyPattern` uses the above specified formulae to compute the parameters and the output, starting from the parameters of the nested workflows. On the other hand, the outputs and parameters of basic activities are evaluated by `EvaluateOut` and `EvaluatePars`, respectively. Finally, the basic and structured algorithm branches are merged, and the results of the evaluation are returned. Regarding the algorithm's complexity, if a is the number of process activities,

and n is the number of possible responses for an activity (correct, latent errors and faults), it is easy to demonstrate that the asymptotic complexity of the `Synthesis` algorithm is $O(n^2 * a)$. In fact, `EvaluateOut`, `EvaluatePars`, and `ApplyPatterns` are $O(n^2)$, because they implement just simple matrix operations. `Synthesis`, on the other hand, implements a visit on the workflow, and has a linear complexity with respect to the number of activities in the workflow $O(a)$. Thus the algorithm is scalable on the number of activities.

6.4 Sensitivity Analysis

Sensitivity analysis techniques have proven to be important tools for understanding and identifying the critical components of a process. These techniques investigate how the uncertainty in the output of a model can be apportioned to different sources of uncertainty in the model's inputs. In our specific context, an architect is primarily interested in identifying the components or the WSs that have a high impact on the overall process reliability or availability, or both. More specifically, the architect is interested in understanding and quantifying the impact that each input (correct or erroneous) has on the process.

One of the most powerful and effective sensitivity analysis techniques is differentiation. It tells us how sensitive a given quantity, which can be expressed as a function of some specific parameters $F(x_1, \dots, x_k, \dots)$, is to these parameters. This technique is achieved by differentiating the quantity function on the considered parameter x_k

$$S_k = \frac{\delta F(x_1, \dots, x_k, \dots)}{\delta x_k}.$$

Once we have obtained the process' result parameters, we can apply sensitivity analysis to investigate specific aspects and contributions. Starting from the Birnbaum importance index [34], and considering a generic property function Y^P , which can be one among C^P , E_j^P , G_j^P , and B_j^P as defined in Table I, the sensitivity of the Y^P property on the i th activity can be specified as

$$S_Y^i = \frac{\delta Y^P}{\delta Y^i}.$$

Applying pivotal decomposition, we can express the i th activity sensitivity on the Y property as

$$\mathbf{S}_Y^i = \frac{\delta Y^P}{\delta Y^i} = \frac{\delta (Y^i Y^P(1^i) + (1 - Y^i) Y^P(0^i))}{\delta Y^i} = Y^P(1^i) - Y^P(0^i).$$

We can therefore evaluate the above formula considering, for the i th component, $\gamma^i = 1$ and $\mathbf{P}^i = \mathbf{I}_n$ in case $Y^P(1^i)$, and $\gamma^i = 0$ for $Y^P(0^i)$.

7 DEPENDABILITY ASSESSMENT IN ACTION

How can we use the proposed technique to obtain valid information from both the user's and the architect's perspectives? The users are mainly interested in selecting the service that implements the needed functionalities, and that best fits their reliability and availability requirements. They want to quantify these parameters, and this can be done using $\langle \gamma^{P^U}, \mathbf{P}^{P^U} \rangle$, as specified above. The architect's goal is to select the WSs to orchestrate based on their reliability and availability. We want to provide useful information, such as the parameters $\langle \gamma^{P^U}, \mathbf{P}^{P^U} \rangle$, and $\langle \gamma^{P^A}, \mathbf{P}^{P^A} \rangle$ that, in the case of the architect, can be manipulated to investigate the impact that each component and each response has on the process. To better clarify how to use the proposed technique, we apply it to the example described in Section 5.

In the workflow of Fig. 5, we identified 3 latent errors, and 5 possible faults. Thus, we represent a generic X process activity using the couple $\langle \gamma^X, \mathbf{P}^X \rangle$, where the propagation matrix \mathbf{P}^X is a 9×9 square block matrix. This way, by applying the proposed approach, we get the overall process formulae detailed in [33].

Once we have obtained the flat workflow, it is analyzed applying the rules and formulae of Section 6. In the evaluation, we base the model parameters that characterize the response probability on real values, which are taken from literature [31], [35] whenever possible. In some cases (e.g., `requestFlightChoice` and `makePayments`) we used real values taken from the experiments performed in [35]. Otherwise, we based these parameters on the statistics presented in [31], which were related to generic WSs, thus characterizing all the other WSs with the same value of γ .

The \mathbf{P}^{P^U} and \mathbf{P}^{P^A} that we obtained from our evaluation are reported in Table V; they only differ in the dimensions, as discussed in Section 6.2. Although the model could be

considered to be quite realistic, we are conscious that, to show the effectiveness of the approach, it is better to assess the method in a real-world environment, and conduct a series of comprehensive experiments. This effort is the first objective of our future work: a full case study experimentation on a real process, in which we identify the process and its WS components, measure and benchmark its reliability and availability properties, and then apply the proposed technique to verify and validate the results.

TABLE V
 $\langle \gamma^P, \mathbf{P}^P \rangle$ of the BPEL example process.

Per- spec- tive	γ^P	\mathbf{P}^P
Architect	0.925	$\begin{bmatrix} 0.584711 & 0.003217 & 0.00214947 & 0.0219415 & 0.131942 & 0.0434513 & 0.139091 & 0.0503504 & 0.0231472 \\ 0.201549 & 0.0234032 & 0.000983154 & 0.007376 & 0.507493 & 0.0181956 & 0.0553862 & 0.0274546 & 0.158159 \\ 0.201798 & 0.00147353 & 0.0230995 & 0.00738442 & 0.508207 & 0.0182301 & 0.0554871 & 0.0275532 & 0.156768 \\ 0.256276 & 0.00218176 & 0.00145807 & 0.0260913 & 0.249526 & 0.11463 & 0.079134 & 0.0573242 & 0.213379 \\ 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. \end{bmatrix}$
User	0.99	$\begin{bmatrix} \mathbf{P}^{PA} & \mathbf{0}_{1 \times n} \\ \mathbf{0}_{n \times 1} & 1 \end{bmatrix}$

TABLE VI
Running example process aggregated parameters.

Error — Fault Id	$C^P E_j^P$	G_j^P	B_j^P	SE_j^P	SF_j^P	LE_j^P	LF_j^P
0	0.540631			0.0252493	0.358733	0.530511	
1	0.00297448	0.121995	0.186355	0.0293679	0.708891	0.00313543	0.130034
2	0.00198743	0.0401756	0.186585	0.0295483	0.70848	0.00216396	0.040367
3	0.0202874	0.128605	0.236957	0.0274898	0.660167	0.0200565	0.126504
4		0.0465546					0.0461966
5		0.0214022					0.0256449

An architect is mainly interested in investigating the process responses, using the metrics of Table I, the example values for which are reported in Table VI. These responses highlight a low process correctness ($C^P \sim 0.54$) that gets worse if we also consider the input ($LE_0^P \sim 0.53$). We can also see that there is a high susceptibility to faults, not only in the case of latent errors (SF_j^P from about 0.66 of error 3 to 0.708 of errors 1 and 2), but also in the case of correct inputs (0.359). To identify which WS has the highest impact on the process, we perform sensitivity analysis, as discussed in Section 6.4. We focus on the two main parameters γ^P and C^P . Table VII shows the importance indexes

we obtained.

TABLE VII
Running example WS importance indexes.

WS — Par	gUI	gFO	gHO	gAO	rFC	rHC	rAC	bFC
γ^P	0.930523	0.756366	0.756366	0.354908	0.717285	0.717285	0.319374	0.679512
C^P	0.571764	0.451097	0.451097	0.211154	0.428246	0.428246	0.190475	0.416454
WS — Par	bHC	bAC	mP	vBA	gC	CUA	rS	
γ^P	0.679512	0.287409	0.521171	0.00555751	0.0428233	0.0209602	0.392564	
C^P	0.416454	0.17666	0.509917	0.00378136	0.0311699	0.0122643	0.229536	

These values tell us that the `getUserInfo` WS is the one with the highest impact on the process. Other important WSs are `makePayments`, and the ones in the *<flow>* concurrent workflow. Some of them have the same sensitivity because they are characterized by the same γ and P values, as reported in [33].

Following the algorithm of Fig. 1, to improve γ^P and C^P , the architect can decide to modify the workflow, select alternative WSs, or both. We assume the architect decides to select alternative WSs. Thus, following the directions provided by the sensitivity analysis, the software architect replaces the `getUserInfo` and `makePayments` WSs with two functionally-equivalent services that are characterized by higher response and lower fault propagation probabilities.

TABLE VIII
 $\langle \gamma^P, P^P \rangle$ of the BPEL example process in the new configuration.

Per- spec- tive	γ^P	P^P
Architect	0.935	$\begin{bmatrix} 0.741565 & 0.00361244 & 0.00242493 & 0.0411729 & 0.0968937 & 0.00973072 & 0.0319764 & 0.0524615 & 0.0201621 \\ 0.326115 & 0.0258343 & 0.00154305 & 0.0175611 & 0.407045 & 0.00738891 & 0.0221701 & 0.0327325 & 0.15961 \\ 0.326446 & 0.00193981 & 0.0253975 & 0.0171725 & 0.407705 & 0.00824158 & 0.0222164 & 0.0328311 & 0.15805 \\ 0.403161 & 0.00270911 & 0.00181959 & 0.038958 & 0.217619 & 0.0822477 & 0.0362229 & 0.0626021 & 0.154661 \\ 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. \end{bmatrix}$
User	0.99	$\begin{bmatrix} PP^{PA} & 0_{1 \times n} \\ 0_{n \times 1} & 1 \end{bmatrix}$

The new configuration is thus evaluated to quantify the impact the new WSs have on the process' parameters, as reported in Table VIII. Now $\gamma^P = 0.934822$, and $C^P = 0.693232$; both parameters have been improved. If the architect is satisfied, the process

is deployed; otherwise, the algorithm is reiterated until a satisfactory configuration is identified.

8 DISCUSSION

After presenting the approach and experimenting with it, we discuss its limitations and threats to validity.

Limitations - In Section 4, we presented six basic assumptions that we make in our work. Here we will identify and evaluate the impact that these assumptions may have on our proposed methodology.

The first three assumptions have to do with how we model the business process. They state that container and inner service time-to-failures are statistically independent, that the inner service time-to-response and time-to-failure are statistically independent, and that branching activities can be probabilistically represented. These are quite typical assumptions, and they are very common in literature [3], [4], [20], [21], [24], [25], [35]. The probabilities of executing conditional and loop constructs can be estimated by applying program analysis techniques [36], [37], and by statically inspecting the BPEL source code. Moreover, our experience with real systems [11], [13], [17], [18], [35] have allowed us to assess that these assumptions are acceptable, and do not undermine the quality of our model.

The fourth assumption regards the fact that input and output types should always be distinguishable. Once again, this assumption is not about the business process itself, but about how we create the business process' model, and specifically about how to characterize and classify inputs and outputs. Indeed, we need to be able to uniquely assign each and every input and output type to a specific type category. In other words, the latent errors and faults that can be taken into account by the proposed technique should be univocally detectable from the output they produce. Therefore, it is impossible for two different latent errors or faults to produce the same output.

The fifth assumption states that a BPEL loop should adopt the same input in each iteration. For example, we can have a data structure that aggregates multiple sub-data, and then have each iteration of the loop's body be responsible for manipulating one of

the sub-data. We cannot have a loop in which the outputs of an iteration i become the inputs of iteration $i + 1$.

Although this assumption is about the business process's control- and data-flows, it is actually quite realistic. Having such a loop construct would be quite common if our goal were to develop an iterative algorithm. However, this condition is usually not the case in business processes, given their high-level of abstraction. Instead, business processes mostly use loops to perform a certain task multiple times on different data, which can easily be aggregated to form a single input [38].

Finally, the sixth assumption regarded the fact that the number of parallel activations of an event handler needed to be finite. Once again, this is an approximation we make on the model. Should the software architect find that the fixed limit is undermining the analysis, she can easily increase the limit, knowing that this will not impact the algorithm's performance.

Threats to Validity - Here we follow [39], where four kinds of threats to validity are mentioned: construct, internal, conclusion, and external.

Regarding *construct* and *internal* validity, our goal is to define an approach for the reliability-availability analysis of Web service compositions, orchestrated via the BPEL workflow language. In this type of research, a frequent problem is the accuracy with which the model represents the system. To this end, we used a workflow model, which is the common way to reason about software qualities in Web services [25]. Some problems that are shared with all architectural approaches are, for example, the possible lack of knowledge about the real execution environment, and consequently the difficulty in defining architecture parameters [40], [41]. Some methods have been defined in the literature, mainly based on estimations backed up by measures taken from actual software or from similar applications, and on estimations backed up by experience [26], [40], [41], [42]. In our work, we used as much data as possible from real systems, to defend our underlying assumptions and the values that the factor levels can take.

Regarding *conclusion* and *external* validity, instead of using a real system, which would have been needed to support the latter, we have considered a non-trivial example. However, to make the approach as general as possible we based most of our assumptions on real data.

9 CONCLUSIONS

We have presented a method to analyze the reliability and availability of BPEL processes that need to compose third-party services, which are characterized by multiple failure modes, latent errors, and propagation effects. The approach also fully takes into account the more advanced tools that the BPEL standard gives us to build reliable processes, such as fault, compensation, termination, and event handlers. The method can be seen as a tool that both architects and users can adopt, the former to reason about a process' reliability and availability, especially in the early development stages, and the latter as support for WS selection. We also fully developed a non-trivial case study in the area of travel management, to illustrate the applicability and effectiveness of our approach.

The proposed technique can be improved along several directions. As we observed, parallel compositions can have a variety of application-dependent semantics. We intend to explore current industrial practices to identify the composition patterns that are relevant in practice, and to provide a formal specification of their reliability attributes. Furthermore, we plan to explore the impact of embedding time dependency in the response probability function, relaxing the assumptions of Section 4. This development would help us deal with timeouts, and allow us to automatically synthesize join synchronization points that depend on the parallel branches' response times. Finally, we are currently improving the implementation of our methodology in the context of a real-world testbed, to assess its effectiveness through a more comprehensive set of experiments.

ACKNOWLEDGEMENT

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (<http://www.erc-smscom.org>).

REFERENCES

- [1] OASIS Committee Specification 01, *Reference Architecture Foundation for Service Oriented Architecture Version 1.0*, OASIS, <http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html>, December 2012.
- [2] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Golland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu, "Web services business process execution language version 2.0," OASIS Committee Draft, May 2006.

- [3] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *Component-Based Software Engineering*, ser. LNCS, L. Grunske, R. Reussner, and F. Plasil, Eds. Springer Berlin / Heidelberg, 2010, vol. 6092, pp. 1–20.
- [4] S. Distefano, A. Filieri, C. Ghezzi, and R. Mirandola, "A compositional method for reliability analysis of workflows affected by multiple failure modes," in *Proc. of the 14th Int. ACM Sigsoft Symp. on Component Based Software Engineering, CBSE 2011, Comparch '11*, Boulder, CO, USA, June 20–24. ACM, 2011, pp. 149–158.
- [5] M. R. Lyu, *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill Book Company, 1999.
- [6] R. Hamadi, B. Benatallah, and B. Medjahed, "Self-adapting recovery nets for policy-driven exception handling in business processes," *Distrib. Parallel Databases*, vol. 23, no. 1, pp. 1–44, Feb. 2008.
- [7] S. Modafferi, E. Mussi, and B. Pernici, "Sh-bpel: a self-healing plug-in for ws-bpel engines," in *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, ser. MW4SOC '06. ACM, 2006, pp. 48–53.
- [8] M. Kaniche, K. Kanoun, and M. Martinello, "A user-perceived availability evaluation of a web based travel agency," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, june 2003, pp. 709 – 718.
- [9] S. Gokhale and J. Lu, "Performance and availability analysis of an e-commerce site," in *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, vol. 1, sept. 2006, pp. 495 –502.
- [10] D. Wang and K. S. Trivedi, "Modeling user-perceived service availability," in *Proceedings of the Second international conference on Service Availability*, ser. ISAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 107–122. [Online]. Available: http://dx.doi.org/10.1007/11560333_10
- [11] M. Merzbacher and D. Patterson, "Measuring end-user availability on the web: practical experience," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, pp. 473 – 477.
- [12] N. Sato and K. Trivedi, "Accurate and efficient stochastic reliability analysis of composite services using their compact markov reward model representations," in *IEEE Int. Conf. on Services Computing, 2007. SCC 2007.*, july 2007, pp. 114 –121.
- [13] K. Goseva-Popstojanova, A. D. Singh, S. Mazimdar, and F. Li, "Empirical characterization of session-based workload and reliability for web servers," *Empirical Software Engineering*, vol. 11, no. 1, pp. 71–117, 2006.
- [14] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction of service-oriented systems," in *Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering*, ser. ICSE '10, vol. 1. New York, NY, USA: ACM, 2010, pp. 35–44.
- [15] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Perform. Eval.*, vol. 45, no. 2-3, pp. 179–204, 2001.
- [16] A. Immonen and E. Niemel, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Software and Systems Modeling*, vol. 7, no. 1, pp. 49–65, 2008.
- [17] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large empirical case study of architecture-based software reliability," in *ISSRE*. IEEE Computer Society, 2005, pp. 43–52.
- [18] K. Goseva-Popstojanova, M. Hamill, and X. Wang, "Adequacy, accuracy, scalability, and uncertainty of architecture-based software reliability: Lessons learned from large empirical case studies," in *ISSRE*. IEEE Computer Society, 2006, pp. 197–203.
- [19] R. C. Cheung, "A user-oriented software reliability model," *IEEE Tr. Sw. Eng.*, vol. 6, no. 2, pp. 118–125, 1980.

- [20] R. Reussner, H. W. Schmidt, and I. Poernomo, "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66, no. 3, pp. 241–252, 2003.
- [21] V. Grassi, "Architecture-based reliability prediction for service-oriented computing," in *Architecting Dependable Systems III*, ser. Lecture Notes in Computer Science, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Springer Berlin / Heidelberg, 2005, vol. 3549, pp. 279–299, 10.1007/11556169_13. [Online]. Available: http://dx.doi.org/10.1007/11556169_13
- [22] B. Beizer, *Micro-Analysis of Computer System Performance*. New York, NY, USA: John Wiley & Sons, Inc., 1978.
- [23] E. Hahn, H. Hermanns, and L. Zhang, "Probabilistic reachability for parametric markov models," in *Model Checking Software*, ser. Lecture Notes in Computer Science, C. Pasareanu, Ed. Springer Berlin / Heidelberg, 2009, vol. 5578, pp. 88–106, 10.1007/978-3-642-02652-2_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02652-2_10
- [24] A. Cardoso, "Quality of service and semantic composition of workflows," Ph.D. dissertation, Graduate School of the University of Georgia, Athens, Georgia, August 2002.
- [25] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 3, pp. 281 – 308, 2004.
- [26] S. S. Gokhale and K. S. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," in *ISSRE*. IEEE Computer Society, 2002, pp. 64–78.
- [27] V. Cortellessa and V. Grassi, "A modeling approach to analyze the impact of error propagation on reliability of component-based systems," *LNCS*, vol. 4608, p. 140, 2007.
- [28] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Trans. Software Eng.*, vol. 18, no. 8, pp. 717–727, 1992.
- [29] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, pp. 11–33, January 2004.
- [30] Z. Zheng, Y. Zhang, and M. Lyu, "Distributed qos evaluation for real-world web services," in *Web Services (ICWS), 2010 IEEE International Conference on*, july 2010, pp. 83 –90.
- [31] Z. Zheng and M. R. Lyu, *QoS Management of Web Services*, ser. Advanced Topics in Science and Technology in China. Springer, 2013.
- [32] H. A. Gabbar, "Improved qualitative fault propagation analysis," *Journal of Loss Prevention in the Process Industries*, vol. 20, no. 3, pp. 260 – 270, 2007. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0950423007000411>
- [33] S. Distefano, C. Ghezzi, S. Guinea, and R. Mirandola, "Dependability assessment of web service orchestrations - full version with appendices," Politecnico di Milano, Available on request, 2013.
- [34] F. Hwang, "A hierarchy of importance indices," *Reliability, IEEE Trans. on*, vol. 54, no. 1, pp. 169 – 172, 2005.
- [35] D. Bruneo, S. Distefano, F. Longo, and M. Scarpa, "Stochastic evaluation of qos in service-based systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 10, pp. 2090–2099, 2013.
- [36] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, 2009, pp. 136–146.
- [37] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis, 2nd edn.* Berlin, Heidelberg: Springer-Verlag, 2005.
- [38] R. Khalaf and F. Leymann, "Coordination for fragmented loops and scopes in a distributed business

- process," *Information Systems*, vol. 37, no. 6, pp. 593 – 610, 2012, bPM 2010. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0306437911001104>
- [39] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Apr. 2009.
- [40] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *ICSE. ACM*, 2008, pp. 111–120.
- [41] C. U. Smith and L. G. Williams, *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley, 2002.
- [42] K. Goseva-Popstojanova, A. Mathur, and K. Trivedi, "Comparison of architecture-based software reliability models," in *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, 2001, pp. 22 – 31.

Salvatore Distefano is an Assistant Professor at Politecnico di Milano. His main research interests include non-Markovian modelling, performance and reliability evaluation, dependability, Parallel and Distributed Computing, Cloud, Autonomic, Volunteer, Crowd Computing, Big Data, and Software and Service Engineering. During his research activity, he has contributed in the development of several tools such as WebSPN, ArgoPerformance, and GS3. He has been involved in several national and international research projects. He is an author or co-author of more than 100 scientific papers. He is member of international conference committees, and he is in the editorial boards of several international journals.

Carlo Ghezzi is a Professor and Chair of Software Engineering at Politecnico di Milano, and an Adjunct Professor at the University of Lugano. He is a Fellow of the ACM, Fellow of the IEEE, Member of the Academy of Europe, and Member of the Italian Academy of Sciences. He has been awarded the ACM SIGSOFT Distinguished Service Award. He has been the Editor in Chief of the ACM Trans. on Software Engineering and Methodology, and is currently an Associate Editor of Communications of the ACM, IEEE Trans. on Software Engineering, Science of Computer Programming, Service Oriented Computing and Applications, and Computing. His research has been focusing on software engineering and programming languages. Currently, he is especially interested in methods and tools to improve dependability of adaptable and evolvable distributed applications, such as service-oriented architectures and ubiquitous and pervasive computer applications. He has co-authored over 180 papers, and 8 books; and coordinated several national and international (EU funded) research projects. He has been awarded an Advanced Grant from the European Research Council.

Sam Guinea is an Assistant Professor at Politecnico di Milano. His research mainly focuses on establishing novel techniques and tools for the development of modern autonomic software systems. He is known in the Software Engineering and Service Oriented Architectures research communities for his work on Self-supervising BPEL processes, and the development of the Dynamo BPEL execution framework. More recently, he has shifted his attention to the runtime management of Internet and Cloud-based applications. Sam's research has cumulatively produced more than 50 publications in top-class international journals, conferences, workshops, and books; and his work is highly cited. He has been a member of various program committees for international conferences and workshops.

Raffaella Mirandola is an Associate Professor in the Dipartimento di Elettronica, Informazione e Bioingegneria at Politecnico di Milano. Raffaella's research interests are in the areas of performance and reliability modeling and analysis of software-hardware systems with special emphasis on methods for the automatic generation of performance and reliability models for component based and service based systems, and methods to develop software that is dependable and can easily evolve, possibly self-adapting its behavior. She has published over 90 journal and conference articles on these topics. She served and is currently serving in the program committees of conferences in the research areas, and she is a member of the Editorial Board of the Journal of System and Software, Elsevier. She has been involved in several national and European research projects.