**Adapting web services to maintain QoS even when faults occur**

# Adapting web services to maintain QoS even when faults occur

Marie-Odile Cordier
*University Rennes 1 / IRISA*
*Rennes, France*
*Email: cordier@irisa.fr*

Roberto Micalizio
*University di Torino*
*Torino, Italia*
*Email: micalizio@di.unito.it*

Sophie Robin
*University Rennes 1 / IRISA*
*Rennes, France*
*Email: robin@irisa.fr*

Laurence Roze
*INSA / IRISA*
*Rennes, France*
*Email: roze@insa-rennes.fr*

***Abstract*—The paper addresses the problem of maintaining the quality of service (QoS) of an orchestration of Web services (WS), which can be affected by exogenous events (i.e., faults). The main challenge in dealing with this problem is that typically the service where a failure is detected is not the one where a fault has occurred: faults have cascade effects on the whole orchestration of services. The paper presents a novel methodology to treat the problem that is not based on Web service (re)composition, but on an adaptive re-execution of the original orchestration. Specifically, an orchestrator Manager exploits an abstract representation of the whole orchestration and a diagnostic module to localize the source of the detected failure. Then, the Manager drives the re-execution of the orchestration by deciding which service activities can be skipped, and which others must be re-executed.**

***Keywords*-monitoring, web services, on-line diagnosis, repair, adaptive systems.**

## I. Introduction

Complex Web applications can be defined as a composition of already existing Web services. The composition of Web services has been successfully applied especially in the B2B model in order to support the exchange of services across enterprises and customers [5].

Web services, however, may be affected by a number of faults, which can be both logical (e.g., corrupted data, wrong input data), and hardware (e.g., network faults). These faults typically cause local failures which may propagate in the whole orchestration, leading to a global failure of the application. To make a Web application more flexible and robust to faults, faults must be detected as soon as possible and properly handled. The usual way to deal with this problem is to establish a closed loop of control detecting and reacting to anomalies that might arise during the run of the application itself. This loop is known in literature as the MAPE model [11]: Monitoring, Analysis, Planning and Execution.

There are two main ways to realize the MAPE loop over an orchestration of services: one based on *global composers*, the other based on *local adapters*. In an approach based on global composers, a single MAPE loop involves all the services within the orchestration. A specific service, the orchestrator, monitors and analyses all the other services, and when the monitored conditions suggest a potential misbehavior of a service, the orchestrator substitutes that service with an equivalent one obtaining a new orchestration. On the contrary, in an approach based on local adapters, the MAPE loop is addressed in a purely local way: each service, or even each activity, has the ability to adapt itself. The advantage of global composers is that they propose globally consistent solutions which are usually computationally expensive. On the other hand, local adapters are in general more efficient, but the solutions are in general less relevant from a global point of view, especially they are unable to deal with cascading effects.

In this paper, we propose a different methodology to realize the MAPE loop, which falls amidst the two previous kinds of approaches. Our aim is to trade off between the high flexibility of a local adapter and the best quality of service of a global composer. In particular, we do not allow to change the original orchestration; but we endow each activity with the ability of selecting the best way to get its local goal among a number of alternative *modalities*, which make each activity very flexible and apt to solve part of the adaptation problem locally. Since an activity can locally select the best modality only having a global view of the composition, we exploit a global context, called Road Map, which is public and accessible by each activity within the orchestration. The Road Map maintains relevant pieces of information about the whole orchestration and helps the activities in their local process of adaptation. An orchestration Manager is in charge of managing the Road Map, initializing and keeping it up-to-date. In our methodology, the MAPE loop is neither completely global nor completely local; in fact, while the Monitoring and the Execution activities are solved locally by each activity; the Analysis and the Planning phases are solved at a global level.

The paper is organized as follows. Section II describes the application that motivates and illustrates our work. Section III gives the architecture of our approach, explaining the role of the four modules composing the MAPE loop. Section IV explains how services and activities are enriched to make them context-adaptable. Sections V and VI detail the two main modules for adaptation, namely the Adapter and the Manager modules. Section VII illustrates the approach on the applicative example. Section VIII is dedicated to related work and Section IX concludes and gives some perspectives.

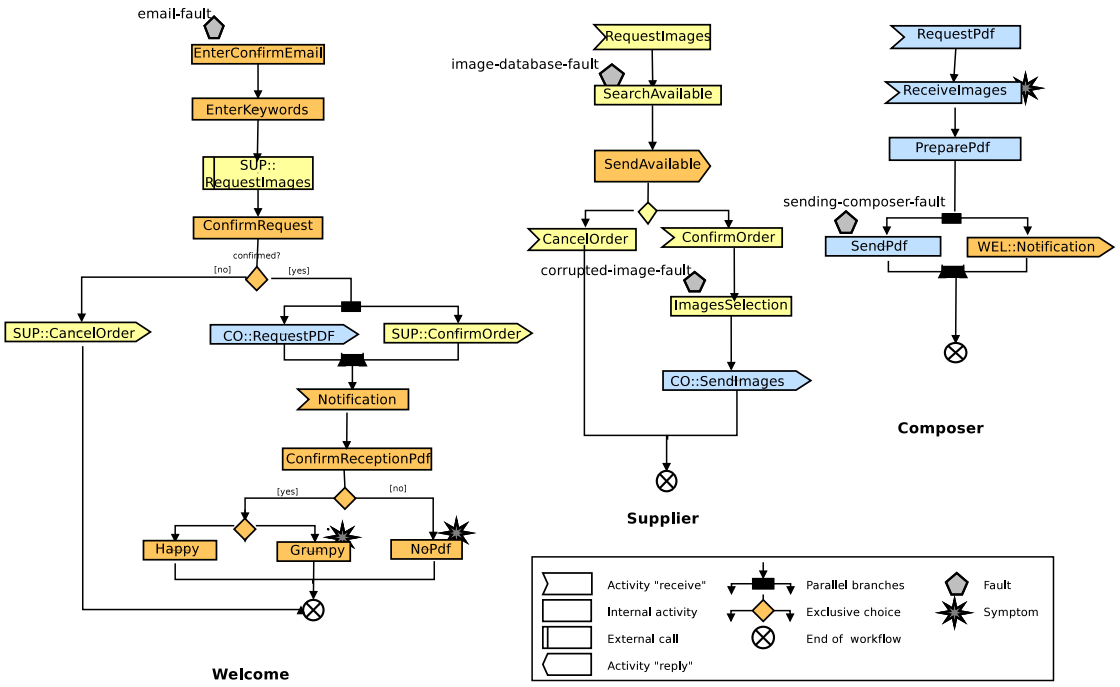


Figure 1. Services, activities and interactions

## II. Application

### A. An illustrative example

We present here the Album Composer example that we use to illustrate and test the methodology we propose. In this example, a customer uses a Web application to compose an electronic album (e.g., a document in pdf format) consisting of a sequence of images (one image by page). The customer gives in input a list of keywords, and gets from the application an album where each image refers to a keyword in the input list. For instance, the customer's keywords could be : {"Mont-Saint-Michel", "George Clooney"}, and the returned album should contain any image representing the Mont-Saint-Michel city and the well-known actor.

To answer the customer's request, the Web application is an orchestration of three services: WELcome, SUPplier, and COmposer. The orchestration of these three services is given in Figure 1. The *WELcome* service gets the customer's request and calls the *SUPplier* service, in charge of finding one image for each keyword, and then asks the customer for confirmation after having displayed the keywords for which images can be provided. In fact, the *SUPplier* may be unable to match some keywords with corresponding images, in that case the customer decides whether (s)he confirms or cancels the command. After the confirmation by the customer[1], the *WELcome* service sends the customer's email address to the *COmposer* service; similarly, the images selected by the *SUPplier* service are sent to the *COmposer* service. This service is in charge of composing the final pdf document and emailing it to the customer, who then confirms to the *WELcome* service the good reception of the pdf file. That terminates the request process by triggering the end of waiting processes for all the services of the orchestration.

[1]Note that the non-confirmation case is not considered in this paper.

### B. Faults and symptoms

Faults may occur during the process of the request by the service orchestration and propagate through the services. In this example, we only consider four faults, one by the *WELcome* service, two by the *SUPplier* and one by the *COmposer*. The first three ones are logical faults and concern erroneous data like bad typing or data base errors. They are: *email-fault*, an error in the email address (the user typed an email address that is not the mail box (s)he is currently working with); *image-database-fault*, an inconsistency between a keyword and the image stored in the supplier database (the "Mont-Saint-Michel" keyword is linked to an image representing a "Chateau-de-la-Loire"); and *corrupted-image-fault*, which occurs when the set of images sent by the supplier to the composer is corrupted and prevent the *COmposer* from performing the composition. The last fault, *sending-composer-fault*, is a hardware fault, and occurs when the *COmposer* is unable, for some technical reasons, to send the pdf file to the customer. Faults are depicted as pentagons in figure1. The observations we rely on to detect the faults are called symptoms. Most of the time, symptoms are observed in a service that is not the one in which the fault occurs. We consider two kinds of symptoms: the first one corresponds to the non-conformity between what is normally expected and what is observed.

In our case, the customer may be unhappy with the pdf file (s)he received, and push the "Grumpy" button instead of the "Happy" one; or the *COmposer* service may be unable to process the images sent by the *SUPplier* because they are corrupted. The second type of symptoms cause abnormal delays in the arrival of messages (i.e., time-outs). This happens when the user does not receive the pdf file within the expected delay. Symptoms are depicted as stars in Figure 1. Faults and symptoms are summarized in Table I.

To preserve the QoS of a customer's request, the faults must be detected, diagnosed and adequate repair actions decided. The example is used to illustrate that the architecture we propose in section III is well-suited to go from symptoms to faults and, thanks to the adapter (section V) and the manager (section VI), to adapt the request process and maintain, as much as possible, the quality of the web service for this request.

## III. Coping with faults by adapting activities

As mentioned in the introduction, any service may be affected by faults that may provoke failures. To cope with this issue, many approaches propose to have a specific service, the orchestrator, that reacts to faults by changing the whole orchestration, the QoS being then preserved by solving a (re-)composition problem. Instead of that, we propose to keep unchanged the orchestration and to adapt the process by means of a Manager module in charge of managing the adaptation loop.

In the rest of this section we first give a general idea about our proposal, and then we provide some details about its internal architecture and compare it with the MAPE loop.

### A. Basic idea

In our approach, all the services within a given orchestration are encapsulated into a Manager module. A customer's request is received by the Manager that "staples" on it what we call a Road Map. The Road Map is the global context the activities use, in conjunction with their own local context, to choose the best way to process the request. The request is then processed by the workflow as usual. When everything goes smoothly, the Manager is in charge of stopping the process and sending a final acknowledgment, that terminates all the service instances in a nominal mode. When a failure is detected, usually by a local monitor, the process is suspended, and the Manager is in charge of finding the best way to pursue the request treatment, according to a new, generally abnormal, context. The Manager is helped in this task by the analysis/diagnosis and the plan adaptation suggested, respectively, by the Analyzer and the Adapter. The idea is to update the Road Map stapled to the request, and then reprocess the request in an alternative way, best adapted to the current context. Rather than allocating new services instances, the Manager keeps alive the same instances as before, and tries to reuse as far as possible

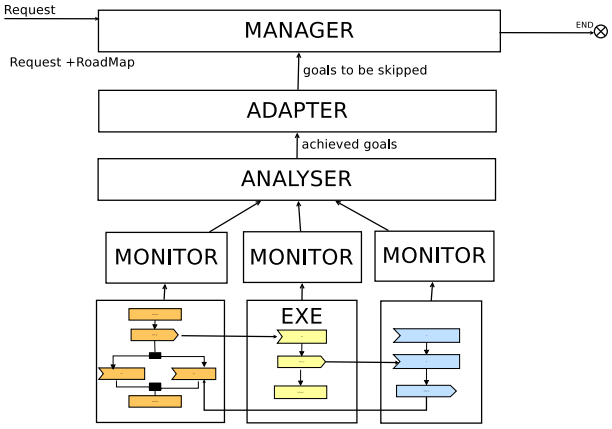


Figure 2. Architecture

data and results previously obtained. The Manager is also in charge of keeping trace of the different attempts and errors encountered during the processing of the request. These data are used by the Manager to determine unrecoverable conditions, and terminate, if needed, the process in a not nominal mode.

### B. Control architecture

Let us now describe the control architecture we propose given in Figure 2. The MAPE loop we propose is composed of four modules in charge of monitoring the execution (MONITOR), analysing it (ANALYSER), looking for possible adaptation changes (ADAPTER) and deciding (MANAGER) what precisely must be done to adapt the execution to the new context. The roles of each of these modules, starting from the lowest level, is now described.

The EXE module corresponds to the workflow, usually composed of interacting distributed services, each service being itself composed of a set of activities, organized according to usual control structures : sequence, conditionals, loops, and so on. The EXE module is in charge of processing the request, both fulfilling QoS constraints and satisfying its goal. In order to make the process adaptive, the activities encapsulate different ways to accomplish their task. These alternative ways are called modalities of the activity. During the execution phase, each activity selects the best way to accomplish its task according to the Road Map associated with the customer's request and the local context of the activity itself.

The Monitor module is a distributed functionality, attached to each service, in charge of logging the actual behavior of that service, detecting when something goes wrong (recognizing a symptom), analyzing local misbehaviors and determining whether it is worth suspending the current process, and entering in a diagnosis/adaptation/re-execution loop. In this case, it awakes the Analyser and sends it monitoring information, called local diagnoses.

| Fault | Occurs in | Observed in | Symptom |
|---|---|---|---|
| *email-fault* | WELcome | WELcome | *time-out*: the user pushed the time-out button; (s)he did not received the file in the expected delay |
| *sending-composer-fault* | COmposer | WELcome | *time-out*: the user pushed the time-out button; (s)he did not receive the file in the expected delay |
| *image-database-fault* | SUPplier | WELcome | *non-conformity* : the user pushed the grumpy button; the images do not correspond to the input keywords |
| *corrupted-image-fault* | SUPplier | COmposer | *non-conformity* : the composer is unable to process the images sent by the Supplier |

Table I
A SUMMARY OF FAULTS AND THEIR SYMPTOMS.

The Analyser (also called Diagnoser [6], [7]) is a centralized functionality in charge of looking for primary causes (global diagnoses) given the local diagnoses provided by the monitors. It is awaken by one of the monitors. It may pool the monitors (initiate a discussion with them) to get a global view of what happened and resulted in a misbehavior. It computes a global diagnosis, as a list of achieved/not achieved goals and transmits such a list to the Adapter.

The Adapter (Planner in the MAPE terminology) is a centralized functionality in charge of looking for the best ways to adapt the process of the request given the diagnosis and the global context. It consists in adapting the current plan, determining which subgoals have to be (re)achieved (and thus which activities must be re-executed), and which subgoals are reusable (and thus which activities may be skipped as no longer needed to achieve the request goal). Of course, a subgoal is reusable when it has been correctly achieved in a run of the application, stored, and fetched in a subsequent run of the application. The Adapter relies on the *composition model*, that is an abstract goal-oriented view of the orchestration that resembles a plan. We suppose the composition model is built by the module in charge of composing the orchestration and transmitted to the Manager.

The Manager is a centralized functionality controlling the life cycle of the Web application. In particular, the Manager initializes and updates the Road Map, and decides when to stop the Web application either in a normal or in an abnormal state.

In conclusion, the loop model we propose matches with the MAPE model; however, not all the four phases are carried out at the same level. The monitoring and execution phases are performed locally by each single service/activity, relying upon the global context associated with the request and possibly a local context that each activity maintains. On the contrary, the analysing/diagnosing and adapting/planning phases are carried out at a global level by the Analyser/Diagnoser and the Adapter modules, respectively. This solution allows the services/activities to take local decision being driven by the global context; and also to ensure a global surveillance of whole workflow when needed. It therefore represents a good balance between the flexibility of local decisions and global consistency.

In the rest of the paper we will focus on two modules of the proposed architecture: the Adapter and the Manager. Details about the Monitors and the Analyser can be found in [6], [7]).

## IV. ENRICHING THE ACTIVITIES

In this section, we describe how the activities are enriched to make them adaptable so that they can take advantages of the Road Map. In our framework, an activity $a$ is characterized by the following set of facets:

- *id*: the identifier of the activity
- *goals*: the list of goals the activity has to achieve
- *modalities* : the set of execution modalities corresponding to different ways of achieving the set of goals
- *policy* : a set of rules of the form "condition : modality"

The idea is that an activity can reach its goals in different manners, called *modalities*. The selection of the most appropriate modality depends on the Road Map and on the policy of the activity itself. In other terms, an activity $a$ can be seen as a nested workflow where the activity policy is realized as a test on both the Road Map (global context) and on the activity context (local context): a modality identifies a branch of the nested workflow (i.e., a sequence of activities) that leads to the activity goal.

For instance, let us consider activity *ImagesSelection* of the Supplier. Activity *ImagesSelection* has a unique goal *images-selected-from-DB*, that is to select images matching the client keywords. It has two modalities, accessing either a *HighQuality* or a *LowQuality* data base. The activity policy is to choose the *LowQuality* one by default (i.e., the first time), and the *HighQuality* data base when it is not the first time the activity is executed. This activity is described by the following frame:

*activity-id* : `SUP::ImagesSelection` ,
*activity-goals* : `images-selected-from-DB`,
*set-of-modalities* :
`HighQualityDBSelection`,
`LowQualityDBSelection`
*policies* :
*if not the first execution of the request* :
`HighQualityDBSelection`
*by default* : `LowQualityDBSelection`

It is easy to see that two subsequent executions of this activity may lead to a different selection of the images as the activity chooses to retrieve images from different DBs.

## V. MODEL-BASED ADAPTATION

The Adapter module is activated whenever the Analyser has discovered some misbehavior during the execution of the services. This means that at least one attempt to accomplish the request has been done, but the occurrence of a fault prevented the achievement of the customer's goals.

The Adapter has the task of inferring the list of *reusable-Goals*. These goals are already available as obtained during a previous run of the Web application; thus there is no need to recompute them again. By exploiting such a list, the Manager can obtain an adaptive, and hence smart, re-execution of the orchestration as only a subset of activities is actually re-executed; all the activities providing goals in *reusableGoals* simply reuse the results they have already inferred. In the rest of the section we discuss the inferences and the models exploited by the Adapter to complete its task.

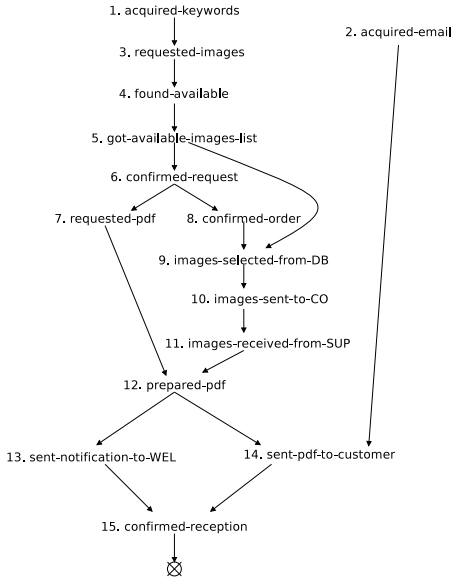


Figure 3. The GD-Graph for the Album Composer scenario.

### A. GD-Graph: the model of the service composition

To infer the *reusableGoals* list, the Adapter exploits an *abstract* model of the composition, which is one of the results of the WS-Composer. In such a composition model, each activity is seen as a goal; input/output variables and data exchanged among activities are abstracted in terms of causal dependencies between goals. The whole composition of services is therefore represented as a network, called *Goal-Dependencies-Graph* (GD-Graph), which is formally defined as a pair $(V, E)$, where $V$ is the set of nodes, each of which corresponds to an intermediate goal, and $E$ is a set of directed edges $\langle g_h, g_k \rangle$: the edge from goal $g_h$ to goal $g_k$ means that $g_h$ is a prerequisite for the achievement of $g_k$, so $g_h$ has to be achieved before $g_k$.

Figure 3 shows the GD-Graph abstracting the services composition for the *Album Composer* scenario. For example, the links between nodes 6 - 7 and 6 - 8 indicate that the goal *confirmed-request* is necessary for achieving both the goals *requested-pdf* and *confirmed-order*; however, these two goals can be achieved independently from each other. Likewise, the goal *prepared-pdf* (node 12) can be obtained only after the achievement of both the goals *requested-pdf* and *images-received-from-SUP* (nodes 7 and 11, respectively).

Table II reports a complete matching between activities and goals in the *Album Composer* scenario.

Note that, the input/output variables of each activity, and which data are actually exchanged among the activities are not traced within the GD-Graph. The main purpose of the GD-Graph abstraction is to capture transversal dependencies among activities through which failures may propagate. Thereby, there is no need to keep neither the notion of which activities belong to which services nor the actual pieces of data exchanged between the activities.

Each goal $g$ in the GD-Graph is associated with a Boolean flag, $avail(g)$, that is *true* when $g$ is storable; that is, when the goal achieved during a run of the system can be stored and re-used during a subsequent re-execution of the services. For instance, physical objects hived in warehouses, or data produced by an activity and stored into the local memory of the same activity are all reusable goals, and hence their *avail* flag is set to true. The flag $avail(g)$ is false otherwise. Table II shows how the availability flag is set for each goal in our running scenario. In principle, goals related to the production of data have the *avail* flag set to *true*; for example, *acquired-keywords* and *acquired-email* are satisfied when, by means of some activities, the customer's list of keywords and his/her email are stored within the local memory of the WELcome service. Goals which represent synchronization points, on the other hand, have the *avail* flag set to *false*, for instance *confirmed-order* and *requested-pdf* are synchronization goals used to start specific workflow activities. The synchronization among activities is strictly related to a particular execution of the services, and hence in case of a re-execution these goals have to be obtained again.

### B. Complementing the GD-Graph

The flag $avail$ states whether a goal has been stored, but it does not specify under which conditions the goal is actually reusable. An erroneous goal should not be reused even if its $avail$ flag is true. For this reason, the GD-Graph is complemented with a Domain Theory (DT), that specifies under which circumstances a goal is reusable. It is the case when either (a) the goal has been correctly achieved[2] and is available, or (b) the goal is no longer needed. A first type of rules in DT corresponds to case (a):

$$\forall g \ (achieved(g) \wedge avail(g) \rightarrow \textit{reusable}(g)).$$

[2]In the following, *achieved* is a short cut for *correctly-achieved.*

| **Service** | **Activity** | **Goal** (node no.) | **Avail** |
|---|---|---|---|
| WELcome | EnterConfirmEmail | acquired-email (1) | true |
| WELcome | EnterKeywords | acquired-keywords (2) | true |
| SUPplier | RequestImages | requested-images (3) | false |
| SUPplier | SearchAvailable | found-available (4) | true |
| SUPplier | SendAvailable | got-available-images-list (5) | true |
| WELcome | ConfirmRequest | confirmed-request (6) | true |
| COmposer | RequestPdf | requested-pdf (7) | false |
| SUPplier | ConfirmOrder | confirmed-order (8) | false |
| SUPplier | ImagesSelection | images-selected-from-DB (9) | true |
| SUPplier | SendImages | images-sent-to-CO (10) | false |
| COmposer | ReceiveImages | images-received-from-SUP (11) | false |
| COmposer | PreparedPdf | prepared-pdf (12) | true |
| COmposer | SendPdf | sent-pdf-to-customer (13) | false |
| WELcome | Notification | sent-notification-to-WEL (14) | false |
| WELcome | ConfirmReceptionPdf | confirmed-reception (15) | false |

Table II
MATCHING ACTIVITIES WITH GOALS.

If a goal has been already correctly achieved during a previous run and is still available, then there is no need to re-achieve it again. For example, let us consider the rule *achieved(acquired-email)* $\wedge$ *avail(acquired-email)* $\rightarrow$ *reusable(acquired-email)*, and let us assume that after the first execution of the services, we infer that *achieved(acquired-email)* holds. Since according to Table II *avail(acquired-email)* is true, we can conclude *reusable(acquired-email)*. This means that during the re-execution phase, the activity responsible for such a goal, here *EnterConfirmEmail*, achieves the goal by retrieving the email address from its local memory, instead of asking it again to the customer.

A second type of rules in DT corresponds to case (b):

$$[\forall g(parent(G, g) \wedge \textit{reusable}(g))] \rightarrow \textit{reusable}(G)$$

where $parent(g_h, g_k)$ is satisfied if the directed edge $\langle g_h, g_k \rangle$ belongs to the GD-Graph. The rule means that, when all the children of a goal $g_h$ are reusable, then the goal $g_h$ itself is reusable. For example, if *prepared-pdf* is reusable, there is no need to execute neither *requested-pdf* nor *images-received-from-SUP* again.

In conclusion, the Adapter exploits the domain theory to determine which goals can be included within the *reusableGoals* list; these goals are either already available or no longer required, and hence the activities providing them may be skipped during a re-execution of the orchestration. The *reusableGoals* list is forwarded to the Manager so that the Road Map is consequently updated and provides a guidance for an efficient re-processing of the whole orchestration.

## VI. THE MANAGER

Once the Adapter has inferred the *reusableGoals* list, it forwards it to the Manager that is in charge of managing the customer's request. The Manager has to decide whether it is worth processing the request again, or it is time to stop the orchestration and return the user a global failure. To make this decision, the Manager may bill the costs gathered so far, both in terms of consumed resources and elapsed time[3]. In case the Manager decides that the request be re-processed, it updates the current Road Map so that the new execution step benefits from what is currently known on the (global) context:

- the *reusableGoals* list gives which goals are no longer required and therefore which activities can be skipped;
- the contextual pieces of information, such as the request's priority or the number of attempts, may be used to select the most appropriate execution modality.

## VII. BACK TO THE ALBUM COMPOSER EXAMPLE

Let us now go back to the *Album Composer* scenario and let us suppose that the composer detects it has received a corrupted image from the supplier. After a Monitor has detected the symptom, the Analyser proposes one diagnosis corresponding to the *corrupted-image-fault* in the *SUPplier* service and a list of correctly *achievedGoals* containing all goals from 1 through 8.

The Adapter is activated with this list as an input and uses the DT rules to infer the *reusable* goals. By using Rule 1, the goals asserted to be *reusable* are {1, 2, 4, 5, 6}. By using Rule 2, the *reusable* property is propagated through the network, and as a consquence also goal 3 is marked as *reusable* even though its *avail* flag is set to false. Finally, the Adapter collects all the goals marked as *reusable* within the *reusableGoals* list and forwards it to the Manager.

Given the *reusableGoals* list, the Manager has now to decide whether the current request should be resubmitted or not. In our current implementation the Manager is always willing to resubmit a request unless a non-recoverable fault has been detected; therefore the Manager updates the Road Map by attaching the *reusableGoals* list and relaunch the whole orchestration.

In the subsequent processing of the request, all the activities producing goals in the *reusableGoals* list are not executed

[3]Aging policies maybe adopted to increase the request's priority over time.

again; in fact, each activity has just to retrieve the results computed at the previous run and forward that results to the subsequent activities. For instance, during the second run, the *EnterConfirmEmail* and *EnterKeywords* activities are skipped: the email and keywords are not (re-)asked to the customer, but they are retrieved from the local memory of the corresponding activities, and forwarded through the workflow. All the activities which are not associated to reusable goals are re-executed as usual; however, they also exploit the Road Map to choose the best way for achieving their goals according to the current context. For instance, when the activity *SUP::ImageSelection* is reactivated, it is known from the Road Map that, in the previous run, a wrong image was selected. Thus, in order to avoid incurring in the same error, an alternative way for achieving the goal is selected. According to its internal policy, *HighQualityDBSelection* is executed instead of *LowQualityDBSelection*, meaning that the requested pictures are retrieved from a different(probably non-defective) DB. If no other faults occur, the user will receive an album where all the images are correct and he/she will terminate the application nominally.

## VIII. Related Work

The paradigm of *autonomic computing* [9] has the objective of developing self-managing systems, which are capable of maintaining an adequate level of QoS while accomplishing their own tasks. Autonomic systems must therefore be endowed with the capabilities of detecting and diagnosing faults, and with the capacity of reacting to faults by self-adapting. Several proposals have been made to cope with this general objective, especially in the case of composed Web services. Composed Web services are encoded in BPEL, which however is not equipped with mechanisms that let Web Services adapt themselves in case of failures. To fill this gap, a number of alternative approaches have been proposed in literature.

A first way is to enhance BPEL with recovery facilities, either by preprocessing the BPEL code and adding recovery activities without having to change the BPEL engine, or by designing dedicated plug-ins. For instance, Charfi et al. propose in [2] a plug-in architecture for self-adaptive web-service composition where self-adaptation features are defined as aspects-based plug-ins. Aspects specify what and how SOAP messages can be modified to add, for instance, security information. The adaptation is done at the messaging layer and this framework requires a BPEL engine adjusted for this specific purpose. O. Ezenwoye and S. M. Sadjadi also present a proxy-based solution to BPEL as a framework for dynamic adaptation of composite Web Services [3]. The TRAP/BPEL framework defines a generic proxy used to encapsulate automatic behavior through the use of self-management policies. This proposal does not require any change of the original code of the BPEL processes nor of BPEL engine, however it is limited as it supports only the substitution of failing partner services. In [8], E. Juhnke and al. identify classes of faults that can be handled automatically and define a policy language to configure automatic recovery behaviors without the need for adding explicit fault handling mechanisms to the BPEL process. The approach provides automatic cloud-based redundancy of services to allow substitution of defective services. S. Subramanian et al. propose in [12] a self-healing policy, called *sh-policy*, consisting of four parts: 1) the *plan* details pre- and post- conditions of each BPEL activity, 2) the *monitor* details the BPEL activities to track during BPEL process, 3) the *diagnosis* details the unexpected failures and their root causes, and 4) the *recovery* details solutions to recover from failures. The plan and monitor parts are identified during BPEL process compilation, while the diagnosis and recovery parts are identified during the BPEL process execution. A Self-heal-BPEL engine extends the classic BPEL engine to support these self-healing mechanisms.

In [1], Baresi et al. exploit probes that monitor the execution of the composition and suggest recovery activities to make the system continue its execution in case of faulty behaviours. The information acquired through the monitors is used to handle the exceptional behaviors. Besides the classical retry and replace strategies, the authors propose a local reorganization of the process. In this case, the BPEL-like process definition is considered as a directed graph and graph transformation rules are applied to modify its topology. This directed graph has similarities with the GD-Graph used by our Adapter to get a global goal-oriented view of the orchestration.

The WS-Diamond framework [13] supports the execution of self-healing web services, thanks to monitoring, diagnosing and recovering functions. Our MAPE architecture is actually inspired by the WS-Diamond architecture. In WS-Diamond, however, the repair process is a global procedure that individuates activities to be compensated and re-tried. In our approach, we take advantage of adaptive BPEL activities and implement the compensate/re-try steps at a local level.

Another point of view, focusing on repair actions acquisition, is developed by B. Pernici and A. M. Rosati [10], who present an approach for learning the repair strategies of Web Services to automatically select repair actions. In particular, a Bayesian classifier is used to drive the repair strategy selection, together with a comparison analysis among faults context, to extract repair actions. Only the two recovery actions retry and substitution are taken into account. Operations provided by the Repair Module are implemented by SH-BPEL, a plug-in for a WS-BPEL engine that allows the execution of repair actions with respect to standard mechanisms.

Finally, G. Friedrich and al. propose a model-based approach to exception handling in service-based processes [4]. A set of repair actions is defined in the process model. Repair is specified as a planning problem whose goal is to build a

plan consisting of recovery actions to be executed when an exception arises during execution.

## IX. Conclusion

This paper addresses the problem of maintaining the quality of service (QoS) of a Web application seen as an orchestration of Web services even when faults occur during the processing of a request. The main challenge is when faults have cascading effects and cause local failures that may propagate in the whole orchestration, leading to a global failure of the application. The usual way to deal with the problem consists in establishing a closed control loop, known in literature as the MAPE model [11]: Monitoring, Analysis, Planning and Execution. The two main trends to implement the MAPE model are either by using *local adapters* - close to exception-handlers, that are efficient but limited to deal with local, non-propagating faults; or by using *global adapters*, that generally solve the problem by calling for a new orchestration where defective activities/services are substituted with alternative ones.

In this paper, we propose a different methodology which falls amidst the two previous kinds of approaches. The aim is to trade-off between the high flexibility of a local adapter and the relevance of a global composer to deal with faults having cascading effects. To reach this objective we propose to enhance the activities within a Web service by viewing them as nested workflows. The different paths inside the workflow of an activity represent the alternative ways with which that activity can reach its expected goals. Thanks to this improvement, each activity has the ability to select the most appropriate way to get a result according to its contextual conditions; thus, we gain the local flexibility we need to cope with faults. On the other hand, a Manager module is in charge of driving the adaptation process and uses a Road Map, stapled to the request, to inform the activities and services of its strategy. The Road Map maintains relevant pieces of information about the whole orchestration and helps the activities in their local process of adaptation.

The proposed approach has several advantages. First of all, the effort of composing an application is done just once. During the life cycle of the application, the orchestration is preserved and there is no need to look for alternative services. This leads to save most of the computational efforts that other recomposition-based solutions have.

In addition, the flexibility achieved with modalities does not require special infrastructures: the activities are still BPEL activities, and modalities are just workflows nested within them.

Finally, although each activity needs to be enriched with modalities, it can respond to anomalies better than a traditional activity since an "enriched" activity can exploit its local context that is preserved from a run of the system to a subsequent one.

We are currently completing an implementation of an e-commerce application, similar to the one illustrated in the paper, to prove the efficiency of the proposed methodology from a practical point fo view.

## References


[1] L. Baresi, C. Ghezzi, and S. Guinea. Towards self-healing service compositions. In *Proceedings of PRISE*, volume 4, pages 11–20, 2004.

[2] A. Charfi, T. Dinkelaker, and M. Mezini. A Plug-in Architecture for Self-Adaptive Web Service Compositions. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '09, pages 35–42, 2009.

[3] O. Ezenwoye and S. M. Sadjadi. TRAP/BPEL: A framework for dynamic adaptation of composite services. In *Proceedings of the International Conference on Web Information Systems and Technologies WEBIST*, pages 1–6, 2007.

[4] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 99:198–215, 2010.

[5] K. D. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[6] X. Le Guillou, M.-O. Cordier, S. Robin, and L. Rozé. Chronicles for on-line diagnosis of distributed systems. In *Proceedings of European Conference of Artificial Intelligence (ECAI)*, pages 194–198, 2008.

[7] X. Le Guillou, M.-O. Cordier, S. Robin, and L. Roze. Monitoring WS-CDL-based choreographies of Web Services. In *Proceedings of the 20th International Workshop on Principles of Diagnosis*, pages 43–50, 2009.

[8] E. Juhnke, T. Dörnemann, and B. Freisleben. Fault-tolerant bpel workflow execution via cloud-aware recovery policies. In *Proceedings of the 35th Conference on Software Engineering and Advanced Applications*, SEAA '09, pages 31–38, 2009.

[9] M. Parashar and S. Hariri. Autonomic computing: An overview. *LNCS*, 3566:247–259, 2005.

[10] B. Pernici and A. M. Rosati. Automatic Learning of Repair Strategies for Web Services. In *Proceedings of the Fifth European Conference on Web Services*, pages 119–128, 2007.

[11] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, 2009.

[12] S. Subramanian, P. Thiran, N. C. Narendra, G. K. Mostefaoui, and Z. Maamar. On the Enhancement of BPEL Engines for Self-Healing Composite Web Services. In *Proceedings of the International Symposium on Applications and the Internet*, pages 33–39, 2008.

[13] WS-Diamond Team. WS-DIAMOND: Web Services, DIAgnosability, MONitoring and Diagnosis. *At your service: an overview of results of projects in the field of service engineering of the IST programme*, chap 9, 2009.