

# Analyzing Impact Factors on Composite Services

Lianne Bodenstaff\*, Andreas Wombacher  
University of Twente  
{l.bodenstaff,a.wombacher}@utwente.nl

Manfred Reichert  
University of Ulm  
manfred.reichert@uni-ulm.de

Michael C. Jaeger  
Siemens AG, Corp. Techn.  
michael.c.jaeger@siemens.com

## Abstract

*Although Web services are intended for short term, ad hoc collaborations, in practice many Web service compositions are offered longterm to customers. While the Web services making up the composition may vary, the structure of the composition is rather fixed. For companies managing such Web service compositions, however, challenges arise which go far beyond simple bilateral contract monitoring. It is not only important to determine whether or not a component (i.e., Web service) in a composition is performing properly, but also to understand what the impact of its performance is on the overall service composition. In this paper we show which challenges emerge and we provide an approach on determining the impact each Web service has on the composition at runtime.*

## 1. Introduction

Regarding the selection of Web services (WS) for a composition, both composition structure and individual characteristics of the services are taken into account when deciding on the most preferable configuration [15]. When managing these compositions at runtime, the focus of existing approaches is on monitoring the quality of service (QoS) provided by each single WS. Typically, composition structure and dependencies between services are not taken into account. However, due to growing complexity of WS compositions, exactly this information is needed to assess composition performance. This situation occurs in many branches of industry besides in a Web service environment, such as electronics companies, car manufacturers, and furniture companies.

Composite service providers struggle to manage these complex constellations. Different services are provided with different quality levels. Services stem from different providers, and have a different impact on the composition. Consider, for example, a service provider who al-

lows financial institutes to check creditworthiness of potential customers. This service is composed of services querying databases, and several payment services (e.g., PayPal and credit card). The composition offered to financial institutes depends on all these services. To meet the Service Level Agreement (SLA) with its customers the company faces the challenge of managing its underlying services. For each SLA violation the company determines its impact on the composition, and it decides how to respond. Generally, complexity of this decision process grows with the number of services being involved in the composition.

The goal of MoDe4SLA [7] is to determine for each service in the composition its *impact* on the composition performance. The latter is measured by analyzing different metrics (e.g. costs and response time) present in SLAs. Furthermore, structural behavior can be enforced by the company or estimated by taking historical performance of service providers into account. Through analysis of both *dependency structure* and *impact* it becomes possible to monitor composition performance taking dependencies between services into account. The advantage of such an analysis is possible identification of *causes* for bad performance.

Fig. 1 depicts the implementation of MoDe4SLA. At design time, we analyze relations between services and the composition with respect to the agreed *response time* and *costs* of the different providers (Step 1 in Fig. 1). The result of this dependency analysis constitutes the input for a subsequent impact analysis (Step 2). During runtime, event logs are analyzed using the event log model, filtering events referring to services and their SLA statements (Step 3). These results, together with impact analysis and dependencies, constitute input for the monitoring interface (Step 4). The latter enables time efficient composition management and maintenance (Step 5).

To support the informal approach presented in [7], this paper introduces formalization of dependency analysis and feedback models. The formalization is supported by a proof-of-concept implementation. The latter provides a means to analyze randomly generated Web service compositions at design time, and to monitor its dependencies and impact during runtime. The innovation of this paper is a

\*This research has been supported by the Dutch Organization for Scientific Research (NWO) under contract number 612.063.409

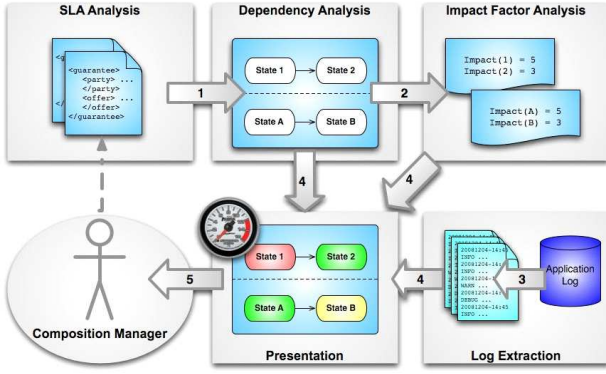


Figure 1. MoDe4SLA Approach

method that processes monitoring data to analyze impact with regard to different (QoS) criteria (i.e., instance-based monitoring) rather than monitoring each service invocation independently.

In Section 2 we describe an informal summary of the contribution after which we give an overview of our approach in Section 3. Sections 4, 5, and 6 describe the formalization process. In Section 7 we present our proof-of-concept implementation. We conclude with related work and a summary in Sections 8 and 9.

## 2. Impact Factors

To determine which service(s) cause SLA violations of the composition, we determine the performance of each individual service and whether this influences the composition SLA, i.e., whether the service has an *impact* on the composition. Each SLA consists of several Service Level Objectives (SLOs). Common SLOs are agreements on response time, availability, and costs. Intuitively, if a service is invoked often and it has a high SLO value then it has a high impact on the composition. In [7] we give an intuitive description on how to combine the number of invocations and realized SLO value by multiplication.

However, this intuitive notion of impact is not sufficient to capture real-life complexity. Consider Fig. 2 representing a service composition where each run invokes two services in parallel: either S1 and S2, or S1 and S3. Intuitively, the expected impact for S1 is  $1 \cdot 10 = 10$  since it is invoked every invocation (i.e., 1) and responds in 10 ms. S2 has an impact of  $0.5 \cdot 20 = 10$  and S3 of  $0.5 \cdot 30 = 15$ , assuming both have 50% chance of being chosen per invocation.

However, in practice structure (i.e., expected number of invocations) and performance (i.e., SLO value) do not suffice to describe the realized impact. It can be expected that most times, S1 finishes before the other service (faster response time). Therefore, S1 does not contribute to overall

response time since this is done by the longer running service. In fact, the *setting* in which the services run (e.g., running parallel with high response time services) should be taken into account as well.

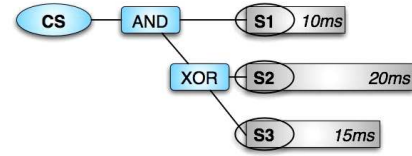


Figure 2. Service Composition Example

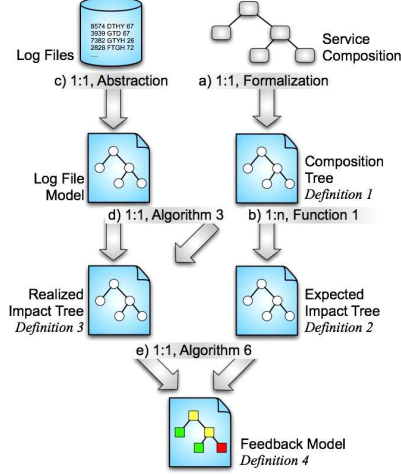
We calculate the impact factor (IF) of each service in the composition as *absolute value* indicating the service *impact share* of the composition for a specific SLO (e.g. response time). By definition, the IFs of the different services add up to 1.0, and are now determined by *structure of the composition* (e.g., parallel running services and estimated number of invocations), *performance of the particular service* (e.g., expected response time), and *performance of other services* in the composition.

## 3. Overall Approach

To automatically derive dependency relations from the composition structure and the SLAs, we propose tree-based formal transformations as depicted in Fig. 3. The composition structure is represented as the *composition tree* (cf. Fig. 3 a). The structure could be derived, for example, from a BPEL process model. Since composition performance is dependent on the performance of the services, we analyze per SLO the expected impact of each service. By combining the structure and the agreed upon level of service, we calculate for each considered SLO at design time the *expected impact tree* (cf. Fig. 3 b). Since agreements are often violated, we monitor at runtime the *realized* impact on the composition for each service. Necessary monitoring information is abstracted from log files (cf. c). Together with the composition this monitoring information is used to calculate the *realized impact tree* (cf. d). The *feedback model* is a performance indicator of the service composition since it shows the difference between expected and realized values (cf. e).

## 4. Design Time

At design time we analyze dependencies of the composition on its underlying services. We determine the expected impact of each service on the overall composition for a specific SLO. A service composition is modelled as a



**Figure 3. Structural Approach**

tree where the top vertex represents the service composition (COMP) and the leafs represent Web services (WS). Connecting vertices and edges depict composition structure. Estimations on the number of invocations are captured by annotating vertices and edges with estimated or agreed upon values. For example, two edges leaving an XOR vertex are annotated with the probability an edge will be chosen. We consider the most commonly used workflow patterns [1] as constructs. Details on vertices are explained in the following. *Composition trees* are defined as follows:

**Definition 1 (Composition Tree)** Let  $\mathcal{V}_s$  be the set of types for service vertices  $\{WS, COMP\}$  and let  $\mathcal{V}_c$  be the set of structural vertices:  $\{AND, ANDDISC, OR, XOR, ORDISC, LOOP, SEQ\}$ . A composition tree is a 6-tuple  $CT(\mathcal{V}_c, \mathcal{V}_s) = (V, E, \rho, \mu, \tau, \sigma)$ , with

- $V$  is a set of vertices,
- $E : V \rightarrow V$  is a set of directed edges,
- $\rho : E \rightarrow \mathbb{R}$  the probability of selection compared to its siblings,
- $\tau : V \rightarrow \mathcal{V}_c \cup \mathcal{V}_s$  specifies the vertex type,
- $\sigma : \{v \in V \mid \tau(v) \in \mathcal{V}_s\} \mapsto \mathbb{R}^n$  specifies the expected SLO values for each SLO, where  $n$  indicates the number of SLOs,<sup>1</sup>
- $\mu : \{v \in V \mid \tau(v) \in \mathcal{V}_c\} \mapsto (\varepsilon \cup \mathbb{R})^3$  annotations of structural vertices are 1) number of started services, 2) number of discriminative success, and 3) number of iterations.

Based on the composition, expected runtime behavior is calculated resulting in an *expected impact tree*. Such a

<sup>1</sup>We assume that all vertices are annotated with the same tuple of SLOs.

Composition	Response Time	Cost
AND	max(total)	sum(total)
AND DISC	max(subset)	sum(subset)
OR	max(subset)	sum(subset)
OR DISC	max(subset)	sum(subset)
XOR	max(one)	sum(one)
Loop	sum(total*)	sum(total*)
Sequence	sum(total)	sum(total)

**Table 1. Vertex Matching between Trees**

tree depicts for an SLO the expected impact of a service per composition invocation. This supports identification of services with high influence on composition behavior. The type of considered SLO determines the transformation algorithm. For example, revisit the example in Fig. 2. Each invoked service has an impact on the composition costs. However, only the slowest responding service influences the composition response time. For each monitored SLO a tree is created (e.g., one for response time and one for costs).

Table 1 shows the types of relations between services per SLO (response time and cost) and construct. Although there exist more SLOs to consider (e.g., availability), this table suffices to demonstrate the principle of our approach. A parallel AND split and AND join (cf. AND in Table 1) succeeds after the slowest responding service finishes (i.e., max(total)), and its total cost is the sum of all invoked services (i.e., sum(total)). For a parallel AND split with discriminative join (AND DISC), a parallel OR split with a discriminative join (OR DISC), and a parallel OR split with normal join, the response time is determined by the slowest invoked service (i.e., max(subset)) and the cost corresponds to the sum (i.e., sum(subset)). For sequences (Sequence) and for sequences executed more than once (Loop) both response time and cost are summed up for all invoked services (sum(total) and sum(total\*), where \* indicates the number of iterations). For an XOR split and join the invoked service contributes to response time and cost (i.e., max(one) and sum(one)). *expected impact trees* are defined as follows:

**Definition 2 (Expected Impact Tree)** Let  $\mathcal{V}_s$  be the set of service vertices  $\{WS, COMP\}$  and let  $\mathcal{V}_i$  be the set of dependency vertices:  $\{max(total), max(subset), max(one), sum(total), sum(subset), sum(one), sum(total*)\}$ . An *expected impact tree* is a 5-tuple  $\mathcal{EIT}(\mathcal{V}_i, \mathcal{V}_s) = (V, E, \rho, \tau, \sigma)$ , where

- $V$  is a set of vertices,
- $E : V \rightarrow V$  is a set of directed edges,
- $\rho : E \rightarrow \mathbb{R}$  the probability of contribution per composition invocation,
- $\tau : V \rightarrow \mathcal{V}_i \cup \mathcal{V}_s$  specifies the type of the vertex,
- $\sigma : \{v \in V \mid \tau(v) = WS\} \mapsto \mathbb{R} \times \mathbb{R}$  Annotations of Web service vertices are in the first dimension

“estimated impact”, and in the second dimension “expected SLO value”,

- $\sigma : \{v \in V \mid \tau(v) = COMP\} \mapsto \mathbb{R}$  annotation of composed services specifies estimated SLO value based on composition structure.

We only introduce algorithms for response time. Due to lack of space, only parts of the algorithm are provided formally (pseudo code), the remaining parts are described verbally. The goal of the expected impact tree is threefold:

```

input :  $v \in V$  & Composition Tree
          $CT = \{V, E, \rho_{CT}, \mu_{CT}, \tau_{CT}, \sigma_{CT}\}$ 
output : Expected average  $rt$  of the composition &
         Expected Impact Tree  $EIT = \{V, E, \rho, \tau, \sigma\}$ 
1 switch  $\tau_{CT}(v)$  do
2   case  $COMP$ 
3      $\tau(v) = COMP$ ;
4      $e_{out} = v \rightarrow v_y \in E$ ;
5      $\rho(e_{out}) = \rho_{CT}(e_{out})$ ;
6      $\sigma(v) = \text{calc}(v_y)$ ;
7     return  $\sigma(v)$ ;
8   case  $AND$ 
9      $\tau(v) = \text{max}(total)$ ;
10     $rt_{max} = 0$ ;
11     $v_{max} = v$ ;
12     $e_{in} = v_y \rightarrow v \in E$ ;
13    foreach  $e_{out} = v \rightarrow v_y \in E$  do
14       $\rho(e_{out}) = \rho(e_{in})$ ;
15       $rt_y = \text{calc}(v_y)$ ;
16      if  $RT_y > RT_{max}$  then
17         $RT_{max} = RT_y$ ;
18         $V_{max} = v_y$ ;
19    foreach  $v \rightarrow v_y \in E \setminus \{v \rightarrow V_{max}\}$  do
20       $\text{reassign}(v_y, 0)$ ;
21    return  $rt_{max}$ ;
22   case  $WS$ 
23      $\tau(v) = WS$ ;
24      $e_{in} = v_y \rightarrow v \in E$ ;
25      $\text{impact}(\sigma(v)) = \rho(e_{in}) \cdot \text{rt}(\sigma_{CT}(v))$ ;
26      $rt(\sigma(v)) = \text{rt}(\sigma_{CT}(v))$ ;
27     return  $rt(\sigma_{CT}(v))$ ;

```

**Function 1**  $\text{calc}(v)$

- Estimate the behavior of the overall composition based on both the contracts (SLAs) with the different service providers and the structure of the composition (i.e., calculate the  $\sigma$ -value).
- Estimate the impact (e.g., on response time) of each subtree on the overall composition (i.e., calculate the  $\rho$ -value for the edges).

- Estimate the impact (e.g., on response time) of each Web service on the overall composition (i.e., calculate the  $\sigma$ -value for the Web service).

All these estimations are based on contracts with the service providers and on the structure of the composition. Both expected QoS and structure determine estimated probability that a service is *invoked*. As described, invocation of a service does not necessarily mean it actually *contributes* to, for example, the response time (e.g., in parallel running branches only the longest running has an impact). Therefore, `Func. calc(v)` determines the probability a service gets invoked and it actually contributes to the overall composition.

```

input :  $v \in V$  and  $z = \rho$  of incoming edge  $v$ 
1  $e_{in} = v_y \rightarrow v \in E$ ;
2  $\rho(e_{in}) = z$ ;
3 switch  $\tau_{CT}(v)$  do
4   case  $XOR$ 
5     foreach  $e_{out} = v \rightarrow v_y \in E$  do
6        $\text{reassign}(v_y, \rho(e_{in}) \cdot \rho_{CT}(e_{out}))$ ;
7   case  $AND$ 
8     foreach  $v \rightarrow v_y \in E$  do  $\text{reassign}(v_y, \rho(e_{in}))$ ;

```

**Function 2**  $\text{reassign}(v, z)$

Vertices  $V$  and edges  $E$  of the expected impact tree are equal to vertices  $V$  and edges  $E$  of the composition tree. The *structure* of both trees is equal, though *naming* of vertices and edges differ. The function traverses recursively through the composition tree, starting with the composition ( $COMP$ ) vertex. Its calculations can be divided in five steps.

Firstly, when traversing the tree the *name of each vertex* ( $\tau$ -value) is determined by its original type (cf. `calc(v)` in line 3, 9, & 22). For example, a parallel split is named `max(total)` in the expected impact tree for response time (cf. Table 1). Secondly, the *probability each edge gets invoked* (cf. `calc(v_y)`,  $\rho$  assignment line 5 & 14) is determined by combining its local probability with the probability its parent edge gets invoked. Now, each Web service “knows” *locally* its probability to be invoked per composition invocation. Thirdly, each vertex determines its *expected average response time* based on the expected response times of its children (i.e., the expected response times of services invoked in that subtree). For example, in `calc(v)` lines 13-15 an AND vertex determines its expected response time based on the maximum response time of all its children. Fourthly, each vertex determines which children branches will have an impact if they are chosen (i.e., *expected contribution*). For example, if an AND vertex has one fast responding child compared to the other children then this child will most likely not contribute to the composition re-

sponse time since it finishes before the rest (cf. `calc(v)`, lines 16-18). Now, each vertex has *global* knowledge on expected behavior of its children. Fifthly, this global information is propagated through the tree, annotating each branch with the *probability it contributes* to the composition per invocation (cf. `Func. reassign(vy, n)` invoked by `calc(v)` in line 19).

The calculation code is not shown for OR split with discriminative join because the code is longer and the general principal has been shown for the AND case (line 8). Informally, for each subset  $s$  of branches from the OR split we calculate likelihood  $l$  of invocation and its response time. Since it is a discriminative join, the expected response time depends on the fastest responding subset  $s'$ . For example, if four out of five branches are started, and three need to finish for the discriminative join to succeed, the theoretical minimum response time can only be the response time of the third-quickest service. Comparable calculations are done for the remaining vertex types.

## 5. Runtime

At runtime we gather monitoring data from log files in a *log file model*. To determine composition performance, we analyze each composition invocation the impact of each service on, for example, response time. Results are represented in the *realized impact tree* where average impact of each service on the composition is depicted.

Log file model  $M$  is an abstraction of log files containing data on composition invocations. Each invocation is represented as a list of invoked Web services for that composition instance. Each element in the list is a 4-tuple containing a time stamp ( $ts$ ), the name of the invoked Web service ( $ws$ ), its costs ( $costs$ ), and its response time ( $rt$ ). It needs to be determined which tuples (i.e., service invocations) in the log file model belong to a specific service composition invocation. We accomplish this correlation by analyzing different time stamps on the service invocations in combination with the service composition structure. A detailed description of this correlation of events is out the scope of this paper (see [6]). The realized impact tree is defined as follows:

**Definition 3 (Realized Impact Tree)** Let  $\mathcal{V}_s$  be the set of service vertices  $\{WS, COMP\}$  and let  $\mathcal{V}_i$  be the set of dependency vertices  $\{max(total), max(subset), max(one), sum(total), sum(subset), sum(one), sum(total^*)\}$ . A realized impact tree is a 5-tuple  $\mathcal{RIT} = (\mathcal{V}_i, \mathcal{V}_s) = (V, E, \rho, \tau, \sigma)$ , where

- $V$  is a set of vertices,
- $E : V \rightarrow V$  is a set of directed edges,
- $\rho : E \rightarrow \mathbb{R}$  the average contribution per composition invocation,
- $\tau : V \rightarrow \mathcal{V}_i \cup \mathcal{V}_s$  specifies the vertex type,

- $\sigma : \{v \in V \mid \tau(v) = WS\} \mapsto \mathbb{R} \times \mathbb{R}$  annotations of Web service vertices are in the first dimension: total contributed SLO value, and in the second dimension: total number of contributions,
- $\sigma : \{v \in V \mid \tau(v) = COMP\} \mapsto \mathbb{R} \times \mathbb{R}$  annotations of the composition node is in the first dimension: total realized SLO value, and in the second dimension: total number of invocations.

The goal of the realized impact tree is threefold:

- Determine realized behavior of the service composition over a specific period of time (i.e., calculate  $\sigma$ ),
- Determine realized impact each subtree has on the overall composition (i.e.,  $\rho$ -value of the edges), and
- Determine realized impact of each Web service on the composition (i.e.,  $\sigma$  of the Web service vertices).

Vertices  $V$  and edges  $E$  of the realized impact tree are equal to the vertices and edges in the composition. Alg. 3 shows implementation for response time. The code is only shown for a subset of vertex types due to lack of space. As argued before, not every Web service invocation eventually contributes to the SLO value of the composition. Therefore, Alg. 3 *analyzes all entries* in the log file model (line 3, Alg. 3) and determines, based on composition structure and other services performance, which entries (i.e., which Web service invocations) impact the composition. For this, recursive `Func. addWS(v)` is invoked in line 5 of Alg. 3. For example, the XOR split determines which children (cf. line 20 of `Func. addWS(v)`) contribute to the overall response time (cf. line 22-26) and returns all contributing Web service invocations in the subtree (cf. `confirm`, line 27). These entries are added to the *confirmed* list (line 6, Alg. 3). Secondly, all confirmed entries are used to *update* total response time and total number of invocations (i.e.,  $\sigma$ -values) of the services (lines 7-10, Alg. 3).

As a last step, Alg. 3 determines contribution per composition invocation (i.e.,  $\rho$ -value) for each edge by invoking recursive `Func. calcImpact(vcomp)` in line 11. This function determines the number of contributions per composition invocation for each edge (i.e., its  $\rho$ -value). Web service leafs calculate  $\rho$ -value of the incoming edge (cf. line 16-19 of `Func. calcImpact`) by comparing number of leaf node invocations with total number of composition invocations. For example, if the composition is invoked 6 times and the leaf node contributes 3 times then it contributes on average 0.5 times to each composition invocation. Each structural node combines information of its outgoing edges and determines the  $\rho$ -value of the incoming edge. For example, in an AND split the overall contribution of the subtree is the summation of  $\rho$ -values of its outgoing edges (cf. lines 11-14).

```

input : Log File Model  $L$  and Composition Tree
           $CT = (V, E, \rho_{CT}, \mu_{CT}, \tau_{CT}, \sigma_{CT})$ 
output : Realized Impact Tree  $EIT = (V, E, \rho, \tau, \sigma)$ 
1  $confirmed = \emptyset$ ;
2  $v_{comp} = \{v \mid \tau_{CT}(v) = COMP\}$ ;
3 foreach  $l \in L$  do
4    $initially = l$ ;
5    $(cf, rt, ts) = \text{addWS}(v_{comp})$ ;
6    $confirmed = confirmed \cup cf$ ;
7 foreach  $tuple \in confirmed$  do
8    $v = ws(tuple)$ ;
9    $rt(\sigma(v)) = rt(\sigma(v)) + rt(tuple)$ ;
10   $contribution(\sigma(v)) ++$ ;
11  $\rho = \text{calcImpact}(v_{comp})$ ;

```

**Algorithm 3:** realized impact

## 6. Feedback Model

The feedback model depicts deviations from agreed upon SLA values by comparing estimated and realized impact trees. Colors on edges and vertices are used to visualize these deviations. Currently, red, green, yellow, darkgreen, and colorless are used (i.e.,  $\theta$ -values) but these can be extended or changed in any preferred way. Intuitively, red and yellow represent negative deviations while green and dark-green represent positive deviations.

**Definition 4 (Feedback Model)** Let  $\mathcal{C}$  be the set of colors  $\{\text{red, green, yellow, darkgreen, no\_color}\}$ . A feedback model is a 6-tuple  $\mathcal{FM}(\mathcal{V}_i, \mathcal{V}_s, (C)) = (V, E, \rho, \tau, \sigma, \theta)$ , where

- $V$  is a set of vertices,
- $E : V \rightarrow V$  is a set of directed edges,
- $\rho : E \rightarrow \mathbb{R}$  realized average contribution per composition invocation,
- $\tau : V \rightarrow \mathcal{V}_i \cup \mathcal{V}_s$  specifies vertex type,
- $\sigma : \{v \in V \mid \tau(v) = WS\} \mapsto \mathbb{R}$  specifies realized impact,
- $\theta : E \rightarrow \mathcal{C}$  specifies the deviation between realized and estimated contribution,
- $\theta : \{v \in V \mid \tau(v) = WS\} \mapsto \mathcal{C}$  specifies the deviation between average contributed value and agreed upon SLO value,
- $\theta : \{v \in V \mid \tau(v) = COMP\} \mapsto \mathcal{C}$  specifies the deviation between realized average value and agreed upon SLO value.

The goal of the feedback model is to support management in identifying causes for badly performing compositions. We accomplish this by giving feedback on:

1. Deviation between expected and realized behavior of the composition regarding an SLO (i.e., its  $\theta$ -value),
2. Deviation between expected and realized contribution of each subtree to the composition,
3. Deviation between expected and realized contribution of each Web service in the composition, and

```

input :  $v \in V$ 
output :  $(confirm, rt, ts)$ : the set of contributing tuples
           $confirm$ , with its overall  $rt$ , and the  $ts$  of the first
          started tuple  $\in confirm$ 
1  $confirm = \emptyset$ ;
2  $rt = 0$ ;
3  $ts = Max$ ;
4 switch  $\tau_{CT}(v)$  do
5   case  $COMP$ 
6      $v \rightarrow v_{child} \in E$ ;
7      $(confirm', rt', ts') = \text{addWS}(v_{child})$ ;
8      $rt(\sigma(v)) = rt(\sigma(v)) + rt'$ ;
9      $invoc(\sigma(v)) ++$ ;
10    return  $(confirm', rt', ts')$ ;
11  case  $AND$ 
12    foreach  $v \rightarrow v_{child} \in E$  do
13       $(confirm', rt', ts') = \text{addWS}(v_y)$ ;
14      if  $rt' > rt$  then
15         $rt = rt'$ ;
16         $confirm = confirm'$ ;
17         $ts = ts'$ ;
18    return  $(confirm, rt, ts)$ ;
19  case  $XOR$ 
20    foreach  $v \rightarrow v_{child} \in E$  do
21       $(confirm', rt', ts') = \text{addWS}(v_y)$ ;
22      if  $confirm' \neq \emptyset \wedge ts' < ts$  then
23        if  $confirm \neq \emptyset$  then
24           $initially = initially \cup confirm$ ;
25           $rt = rt'$ ;
26           $confirm = confirm'$ ;
27           $ts = ts'$ ;
28    return  $(confirm, rt, ts)$ ;
29  case  $WS$ 
30    foreach  $tuple \in initially, ws(tuple) = v$  do
31      if  $ts(tuple) < ts$  then
32         $rt = rt(tuple)$ ;
33         $confirm = tuple$ ;
34         $ts = ts(tuple)$ ;
35    if  $confirm \neq \emptyset$  then
36       $initially = initially \setminus confirm$ ;
37      return  $(confirm, rt, ts)$ ;
38  else
39    return  $(\emptyset, 0, 0)$ ;

```

**Function 4**  $\text{addWS}(v)$

```

input :  $v \in V$ 
1  $v_{comp} = v_x \in V, \tau_{CT}(v_x) = COMP;$ 
2  $e_{in} = v_y \rightarrow v \in E;$ 
3 switch  $\tau_{CT}(v)$  do
4   case  $COMP$ 
5      $\tau(v) = COMP;$ 
6      $\rho = \text{calcImpact}(v_{child});$ 
7     return  $\rho;$ 
8   case  $AND$ 
9      $\tau(v) = XOR;$ 
10     $contribution = 0;$ 
11    foreach  $v \rightarrow v_{child} \in E$  do
12       $\rho_{child} = \text{calcImpact}(v_{child});$ 
13       $contribution = contribution + \rho_{child};$ 
14     $\rho(e_{in}) = contribution;$ 
15    return  $\rho(e_{in});$ 
16   case  $WS$ 
17      $\tau(v) = WS;$ 
18      $\rho(e_{in}) = \frac{contribution(\sigma(v))}{invoc(\sigma(v_{comp}))};$ 
19     return  $\rho(e_{in});$ 

```

**Function 5**  $\text{calcImpact}(v)$

4. Realized contribution per invocation of Web services (i.e.,  $\sigma$ -value) and subtrees (i.e.,  $\rho$ -value).

Vertices  $V$  and edges  $E$  of feedback model are equal to vertices  $V$  and edges  $E$  of estimated and realized impact tree. Alg. 6 calculates the feedback model by computing for each Web service vertex what its composition impact is, e.g. line 5, Alg. 6. This depends on number of contributions per composition invocation (i.e.,  $\rho$ -value) and average SLO when invoked (i.e.,  $\sigma$ -value). Assume Web service S1 has an average response of 10 ms, against 20 ms of the composition. If S1 contributes in fifty percent of the composition invocations (i.e.,  $\rho = 0.50$ ) then the impact factor of S1 is  $\frac{10}{20} \cdot 0.50 = 0.25$ . On average S1 determines 25% of the composition response time.

Furthermore, the color of each Web service and composition is determined by invoking  $\text{color}(\text{real}, \text{est})$  in line 6 and 7 of Alg. 6. This function determines deviation between realized and estimated values as depicted in Func. 7. Each edge is annotated with the realized contribution per composition invocation in line 9 and color  $\theta$  is determined by deviation between expected and realized contribution in line 10.

## 7. Implementation

### 7.1. Generation and Execution

The generation of test compositions is performed with an extended version of the SENECA simulation environ-

```

input :  $EIT(V, E, \rho_E, \tau_E, \sigma_E)$  and
          $RIT(V, E, \rho_R, \tau_R, \sigma_R)$ 
output :  $FM(V, E, \rho, \tau, \sigma, \theta)$ 
1  $v_{comp} = v \in V, \tau_E(v) = COMP;$ 
2 foreach  $v \in V$  do
3    $\tau(v) = \tau_R(v);$ 
4   if  $\tau(v) = WS$  then
5      $\sigma(v) = \frac{rt(\sigma_R(v))}{rt(\sigma_R(v_{comp}))} \cdot \rho_R(e_{in});$ 
6      $\theta(v) = \text{color}(\sigma(v), \text{impact}(\sigma_E(v)));$ 
7   if  $\tau(v) = COMP$  then  $\theta(v) =$ 
    $\text{color}(\frac{rt(\sigma_R(v))}{invoc(\sigma_R(v))}, \text{impact}(\sigma_E(v)));$ 
8 foreach  $e \in E$  do
9    $\rho(e) = \rho_R(e);$ 
10   $\theta(e) = \text{color}(\rho(e), \rho_E(e));$ 

```

**Algorithm 6:** feedback model

```

input :  $real$ : realized value,  $est$ : estimated value
output :  $color$ : the color of the edge or vertex
1  $deviation = \frac{real - est}{est} \cdot 100;$ 
2 if  $deviation \geq 10$  then return  $red;$ 
3 if  $10 > deviation \geq 5$  then return  $yellow;$ 
4 if  $5 > deviation \geq -5$  then return  $green;$ 
5 if  $deviation < -5$  then return  $darkgreen;$ 

```

**Function 7**  $\text{color}(real, est)$

ment [10]. This simulation environment implements a) a structural model of service compositions and b) a QoS model for the handling of QoS attributes of the services in the composition. The generation of compositions is performed randomly with particular input parameters. The parameters are the number of services in total and the range of the planned QoS delivered. Then the software generates - by using the standard random number generator of the Java platform - the following two main parts.

Firstly, the *structure of the composition*. At a given point, the generation software decides between placing a service or a composition structure containing services, both with equal probability. By this scheme, compositions can result in a flat sequence of services or contain nested structures forming a more sophisticated execution plan. There are seven basic executions patterns [10] chosen from the workflow patterns by Aalst et al. [1].

Secondly, the *actual QoS delivered by the service*. At the beginning, this function was used to perform the optimisation simulations for optimising the QoS of service compositions [10]. The generation software has set particular QoS attributes for different candidates in order to select the optimal set of services for the composition according to their QoS. For MoDe4SLA, it is assumed that the selection has taken place and accordingly only one (chosen) service with a particular QoS is required. But in addition, the genera-

tor will randomly generate a probability that is taken at run time to decide whether a service should fail or not. By these two elements, the generator builds up a data structure in application memory that allows the environment to simulate the execution of a service composition.

The simulation performs on the generated test service composition with its QoS attributes as described in the section above. The discrete event simulation simulates the pass of a second (this is the unit of the simulated response time SLO), and tracks the progress among the services of the composition. If a service is finished, then the next service is triggered according to the execution plan. Each service implements the simulation to either start or finish the work as planned. In addition, a random function allows to cancel the execution or miss the planned QoS by running for a longer time. The probability that a particular service fails or runs longer is set at generation time. By this scheme, randomly failing services are simulated. The simulation software generates a log output that tracks a failure of the service. This log is used to perform the impact analysis.

## 7.2. Feedback Models

Service developers receive information on the composition performance through the graphical feedback model as generated by the simulator (cf. Fig. 4). The information can be used to evolve the composition by tuning the structure or renegotiate SLAs of the services. A red colored service indicates worse performance than agreed upon in the SLA, while green indicates proper performance of the service. Yellow indicates the service is not performing perfectly but still within the boundaries set by the company, and dark green indicates a service running even better than the company anticipated. Uncolored services indicate that these never contributed to the overall response time in the considered log file. Therefore, their impact is zero. The edges are colored in the same manner, for example, red indicates the edge contributes more often than expected. Fig. 4 indicates that the SLO for response time of the composition is not met (i.e., it is colored red). Bad performance in this case is caused by two factors. Firstly, *Performance of the services*: together, services 1, 3, and 9 have an impact factor (IF) of over 80% and each service responds on average slower than agreed upon (i.e., yellow or red). Secondly, *Structure of the composition*: badly performing services 3 and 9 are contributing more often than expected (i.e., red incoming edges). The impact and contribution factors in combination with the coloring eases the identification of bad performing services with a high impact so that management of compositions with many services becomes more straightforward.

In this case, we can derive that either badly performing services should be replaced or renegotiated (e.g., service 1,

3, and 9), the SLA agreement with the customer has to be tuned down (i.e., offer less performance), or the structure of the composition has to be adapted so that it suits real-life behavior. Furthermore, we can conclude that services 5, 7, 8, and 10 have no impact on the composition and therefore do not need to be considered for a causal analysis of bad performance, saving valuable analysis time.

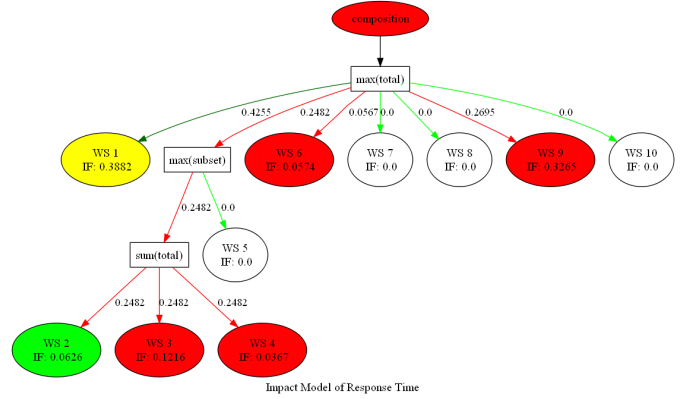


Figure 4. Feedback Model

## 8. Related Work

The work of Casati et al. [13] aims at automated SLA *monitoring* by specifying SLAs and not only considering provider side guarantees but focus also on distributed monitoring, taking the client side into account. Pistore et al. [3] enable run-time monitoring while separating business logic from monitoring functionality. For each instance a monitor is created. Contribution of this approach is the ability to also monitor *classes* of instances, enabling aggregation of impact of several working instances. Smart monitoring [4] implements the monitor functionality itself as service. Three types of monitors handle different aspects of the system. Their approach is developed to monitor contracts with constraints. Further work of Baresi et al. [5] presents an approach to dynamically monitor BPEL processes by adding monitoring rules to them. In contrast, our approach does not require modifications to the process descriptions and thus requires less effort to apply to some application areas. An interesting approach in this direction is work by [11] which, as an exception, does consider the whole state of the system. They aim at monitoring behavioral system derivations. Monitoring requirements are specified in event calculus and evaluated with run-time data. Although many of above mentioned approaches consider services provided by third parties and allow abstraction of results for composite services, none of them addresses how to create this abstrac-



tion in detail. E.g., matching messages from different processes where databases are used are not considered.

Menasce [12] presents response time analysis of composed services to identify impact of slowed down services. The impact on the composition is computed using a Markov chain model. The result is a measure for the overall slow down depending on statistical likelihood of a service not delivering expected response time. As opposed to our approach, Menasce performs at design-time rather than providing an analysis based on analyzing runtime data. In addition, our work provides a framework to cover structures beyond a fork-join arrangement.

A different approach with the same goal is virtual resource manager proposed by Burchard et al. [8]. This approach targets a grid environment where a calculation task is distributed among different grid nodes for individual computation jobs. If a grid node fails to deliver the promised service level, a domain controller first reschedules the job onto a different node within the same domain. If this action fails, the domain controller attempts to query other domain controllers for passing over the computation job. Although the approach covers runtime, it follows a hierarchical autonomous recovery mechanism. MoDe4SLA focusses on identifying causes for correction on the level of business operations rather than on autonomous job scheduling. Further, in *critical path analysis* (CPA) for resource management the preferred order of tasks is determined by analyzing the graph containing all possible paths [14]. However, in CPA each path shows one possible execution while in our analysis the complete graph depicts one execution. Furthermore, in MoDe4SLA we compare estimated with realized behavior which is not done in CPA.

Another research community analyzes *root causes* in services. Dependency models are used to identify causes of violations *within* a company. For example, [2] determine the root cause by using an approach based on dependency graphs. Especially finding the cause of a problem when a service has an SLA with different metrics is a challenging topic. Also [9] uses dependency models for managing internet services, again, with focuss on finding internal causes for problems. MoDe4SLA identifies causes of violations in other services rather than internally. Furthermore, our dependencies between different services are on the same level of abstraction while in root cause analysis one service is evaluated on different levels of abstraction.

## 9. Summary and Outlook

In this paper we describe a formal approach to support management of composite services by analyzing impact factors on service compositions. In continuation of previous work we have now shown the details and feasibility of our approach. The algorithms in pseudo code serve as

a blueprint for our proof-of-concept implementation which supports simulation of service composition runs and the analysis of runtime results. In near future we plan to extend our approach by providing interpretation guidelines for the feedback models to enhance decision making on service composition management. We will use the impact factors to provide reconfiguration suggestions.

## References

- [1] Workflow patterns. <http://www.workflowpatterns.com>.
- [2] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *DSOM*, volume 3278, pages 171–182. Springer, 2004.
- [3] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Runtime monitoring of instances and classes of web service compositions. In *ICWS '06*, pages 63–71, 2006.
- [4] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04*, pages 193–202, New York, NY, USA, 2004.
- [5] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC*, volume 3826, pages 269–282. Springer-Verlag, 2005.
- [6] L. Bodestaff, A. Wombacher, and M. Reichert. On formal consistency between value and coordination models. Technical Report TR-CTIT-07-91, Univ. of Twente.
- [7] L. Bodestaff, A. Wombacher, M. Reichert, and M. C. Jaeger. Monitoring dependencies for SLAs: The MoDe4SLA approach. In *SCC 2008*, pages 21–29, 2008.
- [8] L.-O. Burchard, M. Hovestadt, O. Kao, A. Keller, and B. Linnert. The virtual resource manager: an architecture for SLA-aware resource management. *Cluster Computing and the Grid, IEEE Int. Symposium on*, 0:126–133, 2004.
- [9] D. Caswell and S. Ramanathan. Using service models for management of internet services. *IEEE Journal on Selected Areas in Communications*, 18(5):686–701, 2000.
- [10] M. C. Jaeger and G. Rojec-Goldmann. Seneca - simulation of algorithms for the selection of web services for compositions. In *TES*, pages 84–97, 2005.
- [11] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS*, pages 257–265, 2005.
- [12] D. A. Menascé. Response-time analysis of composite web services. *IEEE Internet Computing*, 8(1):90–92, 2004.
- [13] A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati. Automated SLA monitoring for web services. In *DSOM '02*, pages 28–41, 2002.
- [14] R. Willis. Critical path-analysis and resource constrained project scheduling-theory and practice. *European Journal of Operational Research*, 21(2):149–155, 1985.
- [15] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.