A Reliability Improvement Method for SOA-Based Applications

Goran Delac, Marin Silic and Sinisa Srbljic, *Senior Member, IEEE*

Abstract—As SOA gains more traction through various implementations, building reliable service compositions remains one of the principal research concerns. Widely researched reliability assurance methods, often rely on applying redundancy or complex optimization strategies that can make them less applicable when it comes to designing service compositions on a larger scale. To address this issue, we propose a design time reliability improvement method that enables selective service composition improvements by focusing on the most reliability-critical workflow components, named weak points. With the aim of detecting most significant weak points, we introduce a method based on a suite of recommendation algorithms that leverage a belief network reliability model. The method is made scalable by using heuristic algorithms that achieve better computational performance at the cost of recommendation accuracy. Although less accurate heuristic algorithms on average require more improvement steps, they can achieve better overall performance in cases when the additional step-wise overhead of applying improvements is low. We confirm the soundness of the proposed solution by performing experiments on data sets of randomly generated service compositions.

Index Terms—Software reliability, modeling and estimation, web-based services

1 Introduction

Service-oriented architecture (SOA) [1] is an architectural style that provides guidelines for development of loosely coupled distributed systems. The concept relies on services, self-contained applications deployed on various network nodes which can be consumed using well-established interfaces. Multiple SOA implementations exist, most prominent being the *Web Services* (*WS*) [2], [3] technology. However, other SOA implementations can be expected to gain more traction in the coming years with the continuous proliferation of *cloud computing* and increasing popularity of *software as a service* (SaaS) platforms [4], [5]. One of the most pronounced benefits of SOA are service compositions, component-based applications built by combining the existing services. The concept of compositions makes SOA particularly popular in designing a large variety of systems that benefit from clear separation of interests. For instance, when designing enterprise systems, different segments of functionality within a business process can be developed independently by different organizational units. However, designing service compositions also presents additional challenges as services can be deployed by third parties over which the composition developer has no supervision. A strong concern in such an environment is the necessity to design a composition with an adequate level of non-functional properties, like reliability, availability or other *Quality of Service* (*QoS*) parameters [6]. In this paper, we focus on reliability estimation and improvement methods for service compositions. Hence, we define reliability as the likelihood that a service (composition) invocation will complete properly, i.e. according to its specifications, potentially in the presence of faults.

A common approach to improve reliability and other *QoS* parameters of a service composition is by dynamic service selection at run time [7], [8], [9], [10], [11], [12], [13]. In a dynamic service composition a set of functionally equivalent services exists for each service invocation and actual services are incorporated into the execution configuration depending on their most recent *QoS* parameters. However, two dominant issues limit the application of dynamic compositions on a larger scale: service selection and detection of equivalent services. Since service selection at run time is bonded by additional constraints, like statefullness and composability [7], state-based reliability models need to be applied. However, such models are prone to state explosions [14], making it difficult to support more complex compositions. The other commonly used approach treats service selection as an optimization problem. However, such optimization problems are complex [15], often *NP-hard*, which limits their use. Moreover, discovering and maintaining sets of functionally equivalent services, i.e. homogeneous *service communities* [8], [16], [15], also proves to be a nontrivial task [17], [18], [19].

We argue that when faced with the above mentioned constraints, service composition developers would benefit from a higher level reliability improvement methodology that would enable them to selectively improve parts of a service composition—those most prone to failures. That way it would be possible to achieve the desired level of service composition reliability while

• *G. Delac, M. Silic and S.Srbljic are with the University of Zagreb, Faculty of Electrical Engineering and Computing, Consumer Computing Laboratory, Unska 3, 10000 Zagreb, Croatia.*
E-mail: {goran.delac, marin.silic, sinisa.srbljic}@fer.hr, {gdelach, marin.silic}@gmail.com

spending less resources. Specifically, we suggest an iterative reliability improvement method which involves: (1) modeling the influence of atomic services on the overall service composition reliability and (2) utilizing the reliability model to recommend the most suitable weak points where the composition is to be strengthened—by either performing service replacement, applying fault tolerance strategies or by dynamic service invocation. Since the proposed approach is particularly useful when constructing service compositions on a larger scale, following challenges need to be addressed: (i) size of the reliability model should scale well with the number of services and the complexity of the workflow, and (ii) weak point recommendation methods should scale well with regard to the model size and the additional improvement step-wise computational overheads, like overheads of discovering equivalent services.

To address challenge (i), we present a service composition reliability model based on belief networks. This model is path-based, meaning that the overall composition reliability is calculated based on multiple influence paths that stem from the workflow structure. Although such an approach avoids state explosions and, thus, offers a more compact and scalable model representation, it is not capable of addressing run time constraints—limiting the proposed method to design time. Furthermore, modeling the impact a certain service has on the overall composition reliability through influence paths makes possible to bind that impact to a single random variable. Thus, the model is easily modifiable as only a single random variable needs to be updated to account for the made reliability improvements.

To address challenge (ii), we present a suite of weak point recommendation algorithms with varying complexity and accuracy. We motivate our approach by the fact that a more accurate, but also more computationally demanding, weak point recommendation algorithm can be less applicable in cases when additional step-wise improvement overhead is low. Thus, it is less resource intensive to improve the composition's reliability by performing more improvement steps that individually yield a lower reliability growth. To that end, we propose three algorithms: *WP-Influence*; and two heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath*. While the *WP-Influence* algorithm recommends the weak points by calculating the exact influence of each service in the composition, the heuristic algorithms are constructed to traverse the model only once, which makes them less accurate but also less computationally demanding.

We confirm the feasibility of the proposed approach by performing an extensive evaluation on data sets of randomly generated reliability models. The results indicate that *WP-Influence* provides most accurate weak point recommendations, while *WP-WeightedPath* is on average more accurate than the *WP-WeakestPath* algorithm. Given the additional step-wise computational overhead, the evaluation results suggest that it is possible to construct a hybrid approach—that conducts the initial composition improvement steps using a more accurate algorithm and the final steps using a less accurate algorithm—to further improve the performance of the reliability improvement method.

The remainder of the paper is organized as follows: Section 2 gives an overview of the proposed reliability improvement method. The reliability model for service compositions is introduced in Section 3. Furthermore, the weak point recommendation algorithms are defined in Section 4. The evaluation of accuracy and performance for weak point recommendation algorithms is presented in Section 5 followed by an overview of the related work in Section 6. Finally, the paper is concluded with discussion and future work prospects in Section 7.

2 Reliability Improvement Method

In this paper, we propose an iterative reliability improvement method for service compositions based on the extension of our previous work in [20]. The method consists of: *reliability estimation*, *weak point recommendation* and *weak point strengthening* steps, as defined by the overview in Fig. 1. In the rest of this section, we briefly describe each of the stated steps.

2.1 Reliability Estimation

To perform reliability estimation, a reliability model for service compositions based on belief networks is applied. The underlying dependencies between input random variables, representing atomic services, and service composition output random variable are formed by observing the workflow structure, as defined in Section 3. To make the model easily modifiable we bound the influence of an atomic service to a single input random variable. Thus, when introducing service-level workflow changes into the model, only a single input variable needs to be modified. In general, composition workflows are defined using *Composition Frameworks* that can support various representations, such as domain specific composition languages like *BPEL* [21], *Yawl* [22], or *SSCL* [23]. In this paper, we represent composition workflows using UML activity diagrams, into which other representations can easily be translated.

Crucial to performing reliability estimation is to know the reliability of each individual atomic service. In this this paper, we do not deal with the issue of attaining atomic service reliability and hence assume that this information is available in the *Service Reliability Registry*. More details on the existing approaches in solving this problem are presented Section 6.

2.2 Weak Point Recommendation

If the estimated composition's reliability does not meet the expected non-functional requirements, further improvement actions are needed. Specifically, in the *weak point recommendation* step, the afore mentioned reliability model is utilized by a set of recommendation algorithms

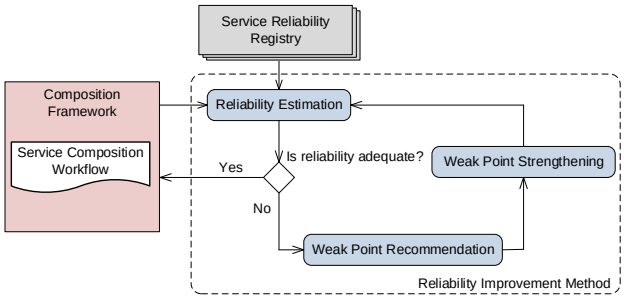


Fig. 1: Reliability Improvement Method Overview.

to compute a list of service composition's weak points, defined as follows:

Definition 1: List of weak points is a list of atomic services ordered by the priority in which they are to be strengthened. Furthermore, an optimal list of weak points is one that orders the atomic services by their exact influence on the composition's overall reliability, i.e. application of this list yields the highest reliability growth.

In this paper we present a set of weak point recommendation algorithms that vary in recommendation accuracy and computational performance. The algorithms are defined in Section 4.1.

2.3 Weak Point Strengthening

In the *weak point strengthening* step, a list of weak points is traversed to find the first entry for which a reliability improvement solution exists. In general, a weak point can be strengthened by performing a simple service replacement, i.e. the service is replaced with a more reliable one, or by introducing fault tolerance strategies that are at developer's disposal (more details in Section 4.2). We consider a valid improvement solution to be the one that has higher reliability than the recommended weak point. If multiple improvement solutions are available, the one with the highest estimated reliability is considered to be the most suitable one. It should be noted that criteria other than reliability can be applied to select an improvement, as discussed in Section 6. Upon modifying a model to incorporate the selected improvement, the overall service composition reliability is recalculated and iterative process of reliability improvement is continued until the desired reliability level is reached or there are no valid reliability improvement solutions left.

2.4 Service Composition Example

Throughout the paper we will be referring to a composition example defined by the UML activity diagram in Fig. 2. The composition workflow is built using atomic services $S_1, ..., S_6$. First, services S_1 and S_2 are invoked sequentially, followed by concurrent execution of services S_3 alongside S_4 and S_5. The workflow is then joined at invocation of service S_2. Finally, one of two execution paths containing services S_1, or S_1 and S_6 is executed depending on the branch condition.

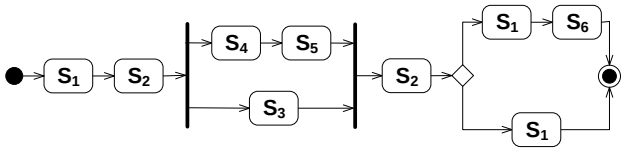


Fig. 2: Example of a service composition.

3 Reliability Model

We introduce a reliability model for service compositions based on the principle of belief networks (Bayesian networks). The model is defined as a directed acyclic graph $M = G(V, E)$, where V is a finite set of vertices representing random variables and E a finite set of edges representing the dependency between random variables. For instance, a directed edge pointing from random variable Y_j to X_i denotes that random variable X_i is conditionally dependent on the random variable Y_j. Each reliability model has multiple input random variables (nodes) representing reliability of atomic services or fault tolerance constructs, and a single output variable denoting the overall service composition reliability. A layer of hidden nodes is constructed between input and output nodes based on the service composition's workflow. We define a set of possible states for each random variable $X \in V$ to be $\Omega = \{\text{normal operation}, \text{fault occurrence}\}$. Furthermore, based on Ω, we define the random variable X_i as follows:

$$X_i(\gamma) = \begin{cases} 1, & \gamma = \text{normal operation} \\ 0, & \gamma = \text{fault occurrence} \end{cases} \quad (1)$$

Since random variables X_i are discrete, we introduce a probability mass function defined for outcomes $e \in X_i(\gamma)$ as follows:

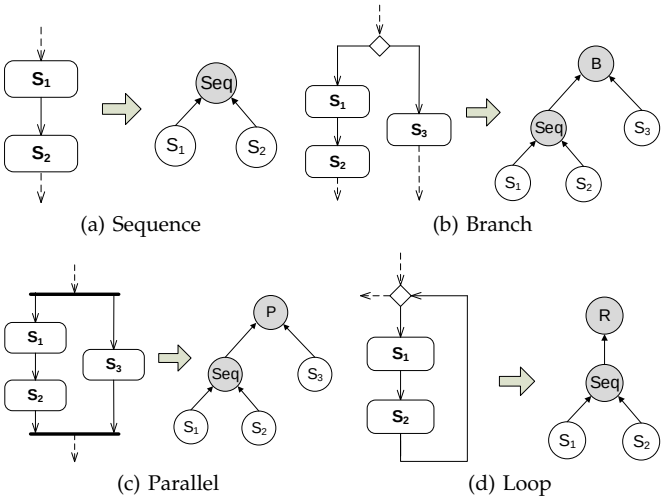
$$\rho_{X_i}(e) = \begin{cases} p(X_i = 1), & e = 1 \\ p(X_i = 0) = 1 - p(X_i = 1), & e = 0 \end{cases} \quad (2)$$

Thus, the probability of an outcome in which a random variable X_i is assigned value 1, $p(X_i = 1)$, is in fact equal to reliability of a composite application, atomic service, or a workflow subset in case X_i is a hidden model node. In the same way, $X_i = 0$ is is regarded as the probability of fault occurrence. For space considerations, the outcomes $X_i = 1$ and $X_i = 0$ will be denoted as X_i and $\neg X_i$, respectively throughout the rest of the paper. A set of all possible outcomes P_O for n random variables is defined as follows:

$$P_O = \{(e_1, e_2, \ldots, e_n)\} \, | X_i = e_i, e_i \in \{0, 1\} \quad (3)$$

where e_i is an outcome assigned to random variable X_i. Since there are 2 possible outcomes for each random variable, a set of all possible outcomes for n random variables contains 2^n elements.

By applying the law of total probability, we can calculate the reliability represented by random variable X_i by summing out all the possible outcomes for random



(a) Sequence (b) Branch

(c) Parallel (d) Loop

Fig. 3: Nodes supported by the reliability model.

variables $Y_1, \ldots, Y_n$. Therefore, $P(X_i)$ can be calculated as follows:

$$P(X_i) = \sum_{j=1}^{2^n} P(X_i|o_j)P(o_j),\ o_j \in P_O \tag{4}$$

where $P(X_i|o_j)$ is conditional dependence of variable X_i on the outcome o_j and $P(o_j)$ the probability that outcome o_j will occur. If random variables $Y_1 \ldots Y_n$ are conditionally independent, the probability $P((e_1, \ldots, e_n))$ is calculated as follows:

$$P((e_1, \ldots, e_n)) = \prod_{i=1}^{n} P(e_i) \tag{5}$$

As stated previously, hidden network nodes specify influence paths from input random variables to the model's single output random variable. Therefore, the model supports several types of random variables used to model various workflow constructs present in the composition's activity diagram, as shown in Fig. 3. In effect, the type of random variable specifies how conditional dependencies to other random variables are defined. Throughout the rest of this section we present types of random variables supported by the model.

3.1 Service

Service nodes are random variables that represent reliability of atomic services used to construct a service composition. More specifically, a random variable $S_{i,j}$ represents the reliability of a method j exposed by a service i. Since we presume that all atomic services are independent entities, they are considered to be conditionally independent in the model. As an implication of this hypothesis, all service nodes in the reliability model have no incoming edges and are considered to be input random variables of the reliability model.

3.2 Composition

Composition node C_i is the single output node for the model and it represents the overall reliability of a composite application. The node has no outgoing arcs and is therefore the only leaf in the network. Conditional probabilities for the variables associated with composition node's incoming edges are calculated the same way as for the sequence node described in the following subsection.

3.3 Sequence

Sequence node is used to model a sequence of events present in the composition's workflow that can in turn comprise of simple service invocations or more complex constructs like parallel workflow paths, branch constructs and loops. An example of a sequence node modeling sequential invocation of two atomic services is presented in Fig. 3a. In order for a sequence of service invocations to be successful, all service invocations have to be completed successfully, i.e. without fault occurrences in any of the sequence sub elements. Therefore, the only outcome in which sequential execution is completed without causing a composition failure is $(e_1 = 1, e_2 = 1, \ldots, e_n = 1)$. Thus, conditional probabilities associated with the sequence node's incoming edges are calculated as follows:

$$P(Seq|o_i) = \begin{cases} 1, & o_i \in P_O|e_1, \ldots, e_n = 1 \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

By plugging (6) into (4) the reliability of a sequence node can be calculated using the following expression:

$$P(Seq) = \prod_{i=1}^{n} P(X_i) \tag{7}$$

where $P(X_i)$ is the reliability an individual workflow construct within a sequence. For example, reliability of a sequence node in Fig. 3a can be calculated as: $P(Seq) = P(S_1)P(S_2)$.

3.4 Loop and Repetition

To model repetitive workflow constructs a repeat node R is used. The repeat node has a single incoming arc for which the conditional probability is calculated as follows:

$$\begin{aligned} &P(R|X_j) = P(X_j)^{k-1} \\ &P(R) = P(X_j)^{k-1}P(X_j) = P(X_j)^k \end{aligned} \tag{8}$$

where k is the expected number of repetitions. The repeat node is used to model both loop constructs, as well as multiple service invocations within a sequence. Specifically, multiple invocations of the same service in a sequence can be modeled by a single repeat node with the parameter k set to the number of invocations. For example, if the service S_1 is invoked 4 times, reliability of the repeat node can be calculated as: $P(R) =$

$P(S_1)^{4-1}P(S_1) = P(S_1)^4$. On the other hand, loops need to be unrolled. This is done so that the loop body is modeled as a sequence of events whose repetition is defined by a child repeat node. It should be noted that in this case parameter k cannot be determined from the composition's workflow, but rather it can be estimated or predicted (e.g. using data collected during past invocations). An example of a reliability model for a loop construct is presented in Fig. 3d. In this case, the loop body consists of sequential service invocations (S_1 and S_2). Therefore, a sequence node is generated with the corresponding repeat node as a child.

3.5 Parallel

A service composition workflow can consist of multiple execution paths that are run concurrently. In this case, a successful composition invocation can depend on fault free completion of a subset of the parallel workflow execution paths. Therefore, the conditional probabilities for a parallel node P_i are calculated as follows:

$$P(P^{(k)}|o_i) = \begin{cases} 1, & E_o \subset o_i | \forall e_j \in E_o, e_j = 1 \wedge |E_o| \geq k \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

where $k \in [1, n]$ indicates how many of the n parallel workflow execution paths have to complete successfully in order for a composition to display fault-free behavior. In other words, at least k of n random variables representing parallel execution paths have to assume value 1 for normal operation. In case $k = n$, all the parallel execution paths must be completed properly to avoid composition failure, effectively making the parallel and sequence nodes equivalent. Thus, the overall reliability for the parallel node can be calculated as follows:

$$P(P^{(k)}) = \sum_{i=1}^{m} P(P^{(k)}|o_i)P(o_i), \quad m = 2^n - \sum_{j=0}^{k-1} \begin{pmatrix} n \\ j \end{pmatrix} \tag{10}$$

where m is the total number of summation factors, i.e the total number of combinations in which k random variables are assigned value 1. An example of a parallel node constructed out of an activity diagram is presented in Fig. 3c. If we assume that at least one branch has to complete successfully for a fault-free service composition behavior ($k = 1$), the overall reliability for the parallel node P is equal to: $P(P) = P(Seq)P(S_3) + P(\neg Seq)P(S_3) + P(Seq)P(\neg S_3)$.

3.6 Branch

Branch constructs introduce uncertainty that a specific workflow path will be executed upon a service composition invocation. Such uncertainty reduces the impact a particular workflow path has on the overall reliability, since it is not executed at each composition invocation. Therefore, each workflow path i of the branch construct is assigned the probability of execution p_i. Similarly like for the repeat node, it is not possible to determine probabilities p_i solely by observing the composition workflow. Instead, p_i can be estimated or predicted using the collected past invocation data. In general, the following condition must hold for the branch construct with n branches: $\sum_{i=1}^{n} p_i = 1$. Therefore, conditional probabilities for a branch node with n incoming edges can be calculated as follows:

$$P(B|o_i) = (1 - \sum_{a \in I} p_a),\ I = \{a | e_a \in o_i \wedge e_a = 0\} \tag{11}$$

In effect, the execution uncertainty is modeled so that the outcomes in which faulty paths exist are multiplied by the probability that the paths in question will not be executed. An example of a branch construct with two incoming edges is presented in Fig. 3b. If p_{Seq} and p_{S_3} are the execution probabilities for nodes Seq and S_3 respectively, the overall reliability for the presented branch node can be calculated as follows: $P(B) = P(Seq)P(S_3) + (1 - p_{Seq})P(\neg Seq)P(S_3) + (1 - p_{S_3})P(Seq)P(\neg S_3)$.

Using the presented model definition, the reliability model for the application example in Section 2 can be constructed as in Fig. 4. The model is a network structure with a single leaf C, representing the overall composition reliability and 6 input nodes standing for services $S_1, ..., S_6$. Since service S_2 is invoked 2 times at each composition execution, it is placed as a child node of a repeat node with $k = 1$. Furthermore, since invocations of S_1 are nested in a branch pattern and 2 sequence patterns, S_1 impacts the overall reliability through 3 influence paths, while the impact of other services is expressed through a single influence path.

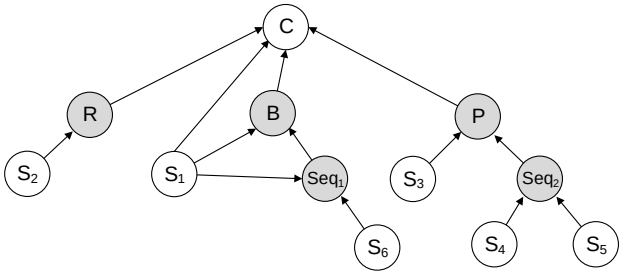


Fig. 4: Reliability model for the composite application example in Fig. 2.

3.7 Atomic Service Dependency

Although when modeling service composition reliability it is common to assume that atomic services are independent of each other [24], this is generally not a valid assumption. However, collecting information on atomic service dependence can prove to be difficult as it is usually available only implicitly in process descriptions and service level agreements (SLA) [25]. Moreover, building a dependency model often requires a deeper insight into the service-oriented system's implementation,

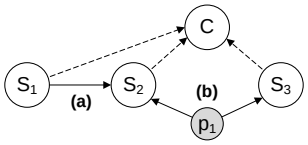


Fig. 5: Atomic service dependency example.

e.g. service deployment details. However, if information on service dependency is available, it can be used to make the presented reliability model more accurate. We observe two basic cases of service dependency, explained by the following examples.

In the first case, service S_1 can be functionally dependent of service S_2 as during its operation S_1 invokes S_2. However, in such a case S_1 cannot be considered an atomic service, but rather a composite one. Thus, S_1 is modeled using a composition node and subsequent constructs (e.g. a sequence node) that describe how S_2 is incorporated into its functionality.

Alternatively, services can be dependent if their inclusion into composition impacts dependability properties of other services. To account for such cases, we extend the reliability model as presented in Fig. 5. In the first case (a), S_2 is modeled to be dependent on S_1, while in the second case (b) S_2 and S_3 are interdependent as they share a certain property, e.g. system resources. For example, including service S_2 into the composition can require reallocation of shared resources, consequently reducing the reliability of service S_3. To avoid model loops, such shared properties are expressed as separate random variables p_i that enable dependency propagation throughout the model. For instance, if an observation that S_2 has changed its reliability is known, the value of p_1 can be recalculated and then in turn reliability of S_3. We do not explicitly state how conditional probabilities linking the S_i and S_j, and S_i and p_i variables are defined as they depend on specific service dependency cases.

4 Service Composition Reliability Improvement

In this section we describe how can the presented probabilistic graphical model be leveraged to recommend weak points using three weak point recommendation algorithms with varying complexity and accuracy. In addition, we discuss how can fault tolerance techniques be incorporated into the model to further strengthen service compositions.

4.1 Weak Point Recommendation

In the following subsections, we present three weak point recommendation algorithms: *Wp-Influence*, *Wp-WeakestPath* and *Wp-WeightedPath*.

4.1.1 WP-Influence Algorithm

Since in the presented reliability model influence of each atomic service is bounded to a single input random variable, the algorithm presented in Fig. 6 (*WP-Influence*) can be used to calculate a list of weak points. At each algorithm iteration, reliability of a tested service, i.e. input variable, is set to 1 (line 5) and the overall reliability is then recalculated (line 6). Based on the achieved increase in service composition's overall reliability, services are ordered into a weak point list so that services whose individual reliability increase caused a greater overall reliability increase are considered to be more suitable improvement candidates, i.e. are placed closer to the beginning of the list (line 8). In effect, the algorithm tests sensitivity to reliability changes of individual atomic services and, thus, yields a highly accurate list of weak points.

```
1:  function WP-Influence(model)          ▷ Returns a list of weak points
2:    wpList ← {}          ▷ List of weak points sorted by reliability growth
3:    for all node in inNodes do          ▷ inNodes - list of input nodes
4:      relTmp ← node.Rel
5:      node.Rel ← 1
6:      finRel ← model.calcRel()          ▷ Returns composition reliability
7:      node.Rel ← relTmp
8:      wpList.insert({finRel, node})          ▷ Inserts (finRel,node) pair
        into the list sorted by finRel
9:    end for
10:   model.calcRel()                          ▷ Resets composition reliability
11:   return wpList
12: end function
```

Fig. 6: *WP-Influence* Algorithm: Recommend a list of weak points by influence.

However, the presented algorithm is not guaranteed to produce an optimal list of weak points at all times. This is due to the fact that the actual reliability growth is impacted by the increase in atomic service reliability achieved as the result of implementing the available improvement solution. For example, a better improvement solution, i.e. one that yields higher overall reliability, can be available for an atomic service that is not recommended as the most significant weak point. Therefore, if known, reliability of the available improvement solutions should be taken into account to make the algorithm more accurate. To that end, we define the *WP-Influence-R* algorithm as a simple extension to the *WP-Influence* algorithm. Specifically, when performing the sensitivity test, rather than by 1, reliability of the tested service is replaced with the actual reliability resulting from the improvement (line 5). That way, the algorithm is guaranteed to produce optimal lists of weak points.

Although the presented algorithms can be leveraged to accurately calculate the influence a given service has on the composition reliability, they are computationally demanding since the overall composition reliability is recalculated once for each input random variable. Specifically, the recursive routine to recalculate the composition reliability needs to be called kn times, where n is the number of services and $k \leq n$ the number of performed service improvement steps.

4.1.2 WP-WeakestPath Algorithm

We argue that the graphical structure of the reliability model can be leveraged to make the weak point rec-

```
1:  function WP-WeakestPath(node)        ▷ Returns a list of weak points
2:    wpList ← {}                                  ▷ List of weak points
3:    if node in inNodes then          ▷ inNodes - list of input nodes
4:      wpList.append(node)
5:      return wpList
6:    else
7:      nextNodes ← node.getChildren()
8:      nextNodes.sortByReliability()
9:      for all cNode in nextNodes do
10:       if cNode not visited then
11:         cNode.markAsVisited()
12:         wpList.append(WP-WeakestPath(cNode) )
13:       end if
14:     end for
15:   end if
16:   return wpList
17: end function
```

Fig. 7: *WP-WeakestPath* Algorithm: Recommend a list of weak points by weakest path.

ommendation process less dependent on the size and complexity of the reliability model at the expense of accuracy. Therefore, we present two heuristic weak point recommendation algorithms that improve computational performance by traversing the model only once. The first algorithm, presented in Fig. 7, named *WP-WeakestPath*, is a greedy approach that detects weak points by traversing the network in a non-causal direction, starting at the leaf node, i.e. the composition node. At each step, the algorithm sorts parent nodes of the currently visited node by their reliability (line 8). A simple invariant holds at each step: the next node to be visited is the node in the parent's list that has not been previously visited and has the lowest reliability (line 10). The search terminates when an atomic service node is reached (line 3). In essence, the algorithm identifies the most suitable improvement candidate by advancing from the composition node towards input random variables while following the path of lowest reliabilities. Although this approach is significantly more lightweight then the previously described *WP-Influence*, it introduces simplifications that make the weak point recommendations less accurate. For instance, the *WP-WeakestPath* algorithm does not take into account that a single input random variable can impact the reliability of a composition through multiple influence paths. An example of such an influence is defined by the reliability model shown in Fig. 4. Specifically, S_1 influences reliability of composition C directly and over nodes B and Seq_1. By ignoring existence of multiple influence paths, the *WP-WeakestPath* can fail to detect the actual most significant atomic services in a service composition.

4.1.3 WP-WeightedPath Algorithm

To reduce the impact of multiple influence paths present in the greedy *WP-WeakestPath* algorithm, we introduce a heuristic algorithm *WP-WeightedPath* presented in Fig. 8. A simple heuristic is used to calculate weights of each path leading from the composition node to input random variables. By summing out all the path weights for a given input random variable, actual influence the variable has on service composition reliability is better observed then it was in case of the *WP-WeakestPath* algorithm. The influence path weights are calculated as follows:

$$w_{i,j} = r_c \prod_{k=1}^{n} (1 - r_k)/r_k \tag{12}$$

where $w_{i,j}$ is the weight of the j^{th} path leading to input random variable i, r_c is the overall service composition reliability and r_k is the reliability of the k^{th} node on the path to variable i. In effect, at each algorithm step the influence reliability of a parent node (r_{pr}) has on the reliability of its child node (r_{ch}) is roughly estimated using this expression: $r_{ch}/r_{pt} - r_{ch}$.

The overall influence of an input random variable is calculated by summing out all the weights for the paths leading to that variable: $inf_i = \sum_j w_{i,j}$. Thus, atomic services with a higher cumulative influence factor inf_i are considered to be better improvement candidates, and the list of weak points is constructed accordingly. The presented approach is on average more accurate and computationally demanding than *WP-WeakestPath* algorithm, but still less accurate than the *WP-Influence* algorithm. This is due to the fact that the heuristic presented in (12) is only a rough estimation of the parent node's influence on its child node.

Similarly like in case of the *WP-Influence* algorithm, if reliability of the available improvement solutions is known, it could be used to make the presented heuristic more accurate. To that end, we presented a simple solution that considers weighting the influence factor inf_i with reliabilities of available improvements. Specifically, the modified influence factor is calculated as $inf'_i = inf_i \cdot r_i$, where r_i is the reliability of available improvement solution for atomic service i. Throughout the rest of the paper, this adaptation is referred to as *WP-WeightedPath-R* algorithm.

4.2 Weak Point Strengthening

The list of weak points calculated utilizing the presented reliability model and recommendation algorithms is used to prioritize improvement actions. An example of the weak point strengthening step for the service composition reliability model in Fig. 4 is given in Fig. 9. Let services $S_1, ..., S_6$ have an equal initial reliability of 0.85 and let each service have a single equivalent replacement of reliability 0.99. Furthermore, assume that parallel workflow paths are redundant, so at least one path has to complete to avoid composition failure ($k = 1$), and that both branches are executed with an equal probability ($p_1 = 0.5$, $p_2 = 0.5$). Since there are 6 possible replacement choices, one for each service, there is a total of 6 improvement steps, as denoted on the x-axis. The overall composition reliability is given on the y-axis of the graph. For the presented example, it can be observed that *WP-Influence* and *WP-WeightedPath* gave more accurate weak point recommendations than the

```
1:  function WP-WeightedPath(node)        ▷ Returns a list of weak points
2:    wpHmap ← {}        ▷ Hash map of weak points: Key - component,
    Value - weight
3:    for all entry in inNodes do          ▷ inNodes - list of input nodes
4:      wpHmap.add(entry, 0.0)
5:    end for
6:    wPaths ← CalculatePathWeights(node, 1.0)
7:    for all entry in wPaths do
8:      weight ← wpHmap.get(entry.node) + entry.weight
9:      wpHmap.update(entry.node, weight)
10:   end for
11:   wpList = wpHmap.generateDescendingSortedList()
12:   return wpList
13: end function
14: function CalculatePathWeights(node, weight)
15:   pathWeights ← {}                     ▷ List of path weights
16:   if node in inNodes then
17:     pathWeights.append({node, weight})
18:     return pathWeights
19:   else
20:     nextNodes ← node.getChildren()
21:     pWeight ← weight * ((1 − node.reliability)/node.reliability)
22:     for all cNode in nextNodes do
23:       pathWeights.append(CalculatePathWeights(cNode,
    pWeight) )
24:     end for
25:   end if
26:   return pathWeights
27: end function
```

Fig. 8: *WP-WeightedPath* Algorithm: Recommend a list of weak points by weighted influence paths.

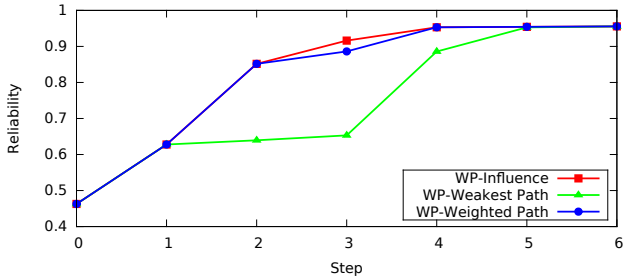


Fig. 9: Reliability improvement results for the composition example in Fig. 4.

WP-WeakestPath algorithm since they both resulted in a higher overall reliability for improvement steps 2-4.

As stated previously, replacing atomic services with more reliable ones is not the only possible improvement solution. Specialized constructs within the execution environment—like service brokers [14]—implementing well-known fault tolerance methods, like *recovery blocks* [26] or *n-version programming* [27] could be introduced into the workflow. Such constructs would facilitate redundancy by invoking several functionally equivalent services to achieve reliability of the improvement solution greater than reliability of each individual equivalent service. That way, greater overall reliability is achieved at the cost of system resources. Multiple strategies on selecting appropriate fault tolerance mechanisms can be applied [15] and we will not discuss them in a grater detail. We briefly describe how can such constructs be incorporated into the presented reliability model. For instance, in the *recovery blocks* approach, services are invoked sequentially and tested until a properly working service has been identified. In such a case, fault tolerance construct is modeled by the RB node with reliability:

$$P(RB_i) = 1 - \prod_{i=1}^{n} (1 - P(S_i)) \tag{13}$$

where $P(S_i)$ is reliability of the i^{th} redundant service and n the total number of redundant services. In effect, $P(RB)$ is equal to probability that event in which all redundant services fail will not occur.

In case of the *n-version programming* approach, services are invoked concurrently and the final result is calculated by a voting algorithm. Thus, the fault tolerant construct based on *n-version* programming is modeled the same way as the parallel node in equation (9). Moreover, parameter k depends on the implementation of the voting algorithm since it can vary for different applications. For instance, if *majority voting* [28] is applied, $k = \lceil (n+1)/2 \rceil$, where n is the number of redundant services.

5 Evaluation

We evaluate the proposed reliability improvement method on an artificially generated data set. The data set consists of 5000 randomly generated service composition reliability models. Each composition is built out of 30 independent atomic services. To assure an appropriate workflow complexity, models have the maximum nesting depth and the maximum number of parents for each node set to 3. At each level, *repetition*, *branch*, *sequence* and *parallel* nodes are generated with an equal probability. The number of repetitions defined by the *repeat* nodes is randomly chosen form the interval $[1, 4]$, while the *parallel* nodes require that at least one concurrent path completes without a fault ($k = 1$). Finally, execution probabilities for all workflow branches defined by the *branch* element are set equally, i.e. probability of execution for a single branch is $1/n$, where n is the total number of branches. In the rest of the section, the described data set is used to analyze accuracy and performance of the weak point recommendation algorithms described in section 4.1. We define the following evaluation measures that are used throughout the section:

Average reliability

The average composition reliability is defined as the arithmetic mean of all compositions' reliabilities in the data set. It is given by the following equation:

$$AvgRel = \frac{\sum_{i=1}^{N} rel_i}{N} \tag{14}$$

where rel_i is the reliability of a service composition i and N is the total number of compositions in the data set.

Average reliability growth

We define the average reliability growth as average reliability increase achieved between two subsequent reliability improvement steps using the following equation:

$$RelGr_k = \frac{\sum_{i=1}^{N} (rel_{i,k} - rel_{i,k-1})}{N} \quad (15)$$

where $rel_{i,k}$ is the reliability of composition i at step k and N is the total number of compositions in the data set.

Average total reliability growth

The average total reliability growth is defined as the average increase in reliability between the composition's initial reliability and reliability achieved after k improvement steps:

$$TotRelGr_k = \frac{\sum_{i=1}^{N} (rel_{i,k} - rel_{i,0})}{N} \quad (16)$$

where $rel_{i,0}$ is the initial reliability of the composite application i, $rel_{i,k}$ the reliability after k improvement steps and N is the total number of compositions in the data set.

Mean absolute percentage error

In order to evaluate the difference in accuracy between recommendation algorithms, mean absolute percentage error measure (*MAPE*) is calculated as follows:

$$MAPE = \frac{\sum_{i=1}^{N} |(x - y)/x|}{N} \quad (17)$$

where x is the value against y is being compared.

5.1 Reliability Growth

One of the key metrics in evaluating the accuracy of weak point recommendation algorithms is reliability growth, i.e. the increase in reliability achieved after strengthening the weak points in the recommended order. We conducted two experiments on the generated data set to assess how the proposed weak point recommendation algorithms behave when atomic services are very similar or equal in reliability and when there is a more significant difference in the services' reliability. In *experiment A*, the reliability of all services was equal and set to 0.99. For each of the 30 atomic services a single replacement service of reliability 0.9999 was available. In *experiment B*, the reliability of the services varied and it was randomly chosen from the interval $[0.7, 0.9]$ with a uniform distribution. Similarly like in *experiment A*, for each atomic service a single replacement service was available. The replacement services had a 20% greater reliability than the ones initially built into the workflow, i.e. they had reliability from the interval $[0.84, 1.0]$. The results for both experiments are shown in Fig. 10.

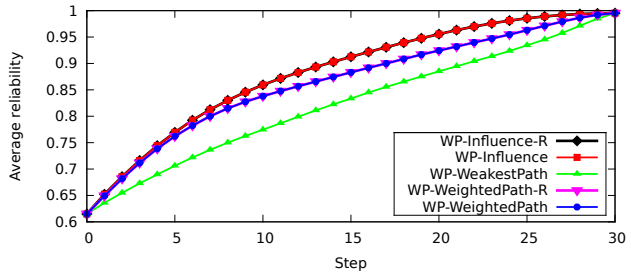
The results for average reliability ($AvgRel$) in both experiments are given by the graphs in Fig. 10a and Fig. 10b. From the graphs it is visible that the *WP-Influence* algorithm achieves higher $AvgRel$ then the heuristic algorithms at each composition improvement step. In addition, it can be concluded that the *WP-WeightedPath* algorithm, which takes into account multiple influence paths, achieves higher $AvgRel$ than the *WP-WeakestPath* algorithm in both experiments. Finally, it can be observed that *WP-Influence-R* and *WP-WeightedPath-R* algorithms expectedly achieve higher $AvgRel$ than *WP-Influence* and *WP-WeightedPath* respectively. To give a more precise insight, *MAPE* is calculated to evaluate accuracy of recommendation algorithms against the optimal solution provided by the *WP-WeightedPath-R*. The results are presented in Table 1. As expected, *WP-Influence* has lower *MAPE* than the rest of the algorithms in both experiments. In fact, in *experiment A*, it achieves several orders of magnitude lower *MAPE* as reliability of replacements are very close to 1. Furthermore, *WP-WeightedPath* path is more accurate than *WP-WeakestPath* as it achieves lower *MAPE*, up to 3.26 times lower in *experiment A*. Finally, it is confirmed that improvements incorporated into *WP-WeightedPath-R* yield higher accuracy, as *MAPE* in *experiment B* is 10.24% lower than for *WP-WeightedPath*. Expectedly, in *experiment A*, *WP-WeightedPath-R* performs the same as *WP-WeightedPath* since all replacement services have the same reliability.

TABLE 1: *MAPE* for difference in $AvgRel$ between recommendation algorithms and *WP-Influence-R*

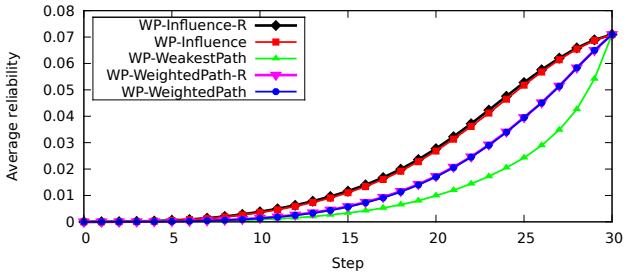
Algorithm	*experiment A*	*experiment B*
WP-Influence	0.000000020392	0.0465564229602
WP-WeakestPath	0.067967853477	0.6233483300111
WP-WeightedPath	0.020843018143	0.4473539879849
WP-WeightedP-R	0.020843018143	0.4365570191153

In order give a better insight into accuracy for the initial improvement steps and to analyze the trend of growth, we present the average reliability growth for both experiments in Fig. 10c and Fig. 10d. When comparing two weak point recommendation algorithms, a better performing algorithm should achieve higher initial average reliability growth and a lower average reliability growth at the final improvement steps. The results presented for both experiments confirm that the *WP-Influence-R* algorithm achieves highest average reliability growth during the initial improvement steps. It can also be concluded that the *WP-WeightedPath* and *WP-WeightedPath-R* algorithms outperform the *WP-WeakestPath* algorithm in both experiments.

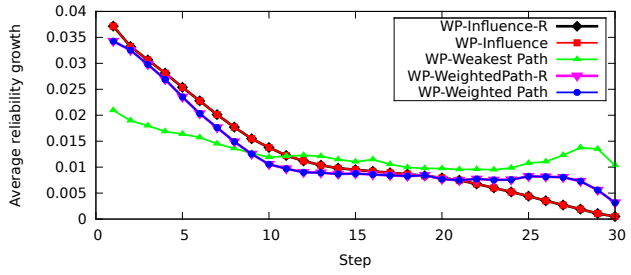
Finally, in Fig. 10e and 10f we show the average number of composition improvement steps required to reach a certain reliability threshold—denoted on the x-axis as the total reliability growth. It should be noted since not all compositions can reach the defined reliability thresholds, average number of steps is calculated for appropriate data set subsets, i.e. only for those applications that can reach the defined threshold. In *experiment A*, the *WP-Influence* algorithm achieves the defined reliability threshold (0.5) while performing 13.45% and 10.01% less improvement steps then algorithms *WP-WeakestPath* and *WP-WeightedPath* respectively. Similarly, in *experiment B*, the *WP-Influence* algorithm reaches the reliability thresh-



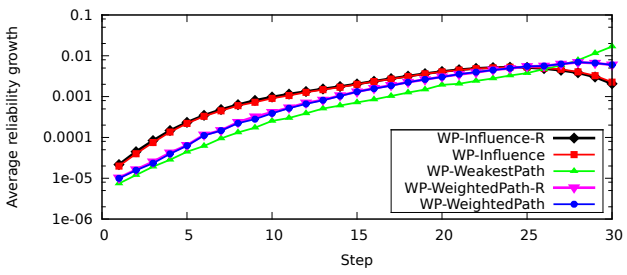
(a) Average reliability: *experiment A*



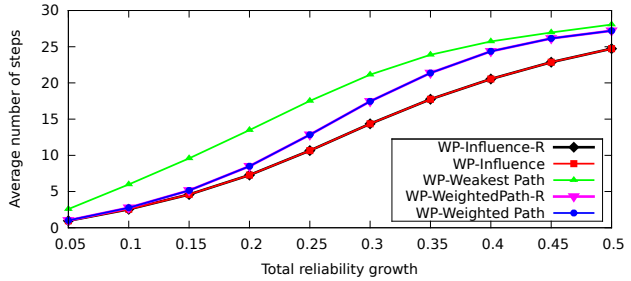
(b) Average reliability: *experiment B*



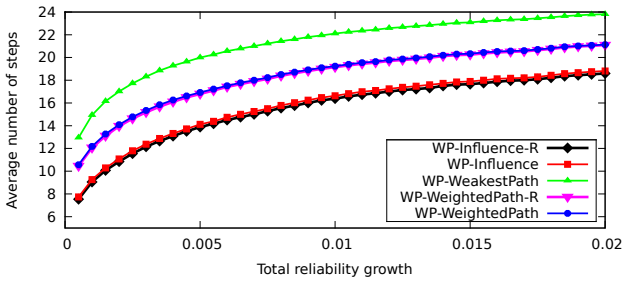
(c) Average reliability growth: *experiment A*



(d) Average reliability growth: *experiment B*



(e) Average number of steps to reach a threshold: *experiment A*



(f) Average number of steps to reach a threshold: *experiment B*

Fig. 10: Reliability improvement results.

old (0.02) in 26.53% and 12.34% less improvement steps then *WP-WeakestPath* and *WP-WeightedPath* respectively. The results expectedly indicate that *WP-Influence-R* and *WP-WeightedPath-R* perform more accurately than their unmodified versions.

5.2 Impact of Atomic Service Dependency

To evaluate the impact of atomic service dependency on weak point recommendation accuracy, we extended the models generated in *experiment A* with additional dependencies. Specifically, 5 service pairs were randomly chosen in each model. In each pair, S_i was made dependent on S_j so that by improving S_j, reliability of S_i would increase 10%. Two experiments were made for each recommendation algorithm. In the first experiment, weak points were recommended using the model that takes into account dependencies (+*D*). On the other hand, in the second experiment, recommendations were performed using the original models that do not take into account dependences (-*D*). In order to evaluate how using (-*D*) models impacts the recommendation accuracy, reliability growth for weak point lists generated in the second experiment was calculated using the extended models (+*D*). The results for $AvgRel$, presented

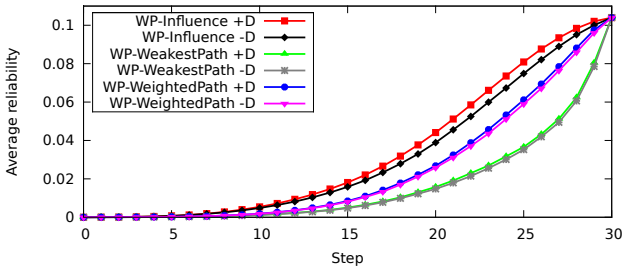


Fig. 11: Impact of atomic service reliability on $AvgRel$.

in Fig. 11, clearly indicate that by ignoring atomic service dependence, the recommendation accuracy decreases. It should also be noted that *WP-Influence* is more susceptible to inaccuracies in (-*D*) models then the heuristic algorithms, as it strongly depends on accurately estimating the influence of each individual atomic service.

5.3 Performance

In order to determine how the size of the model, i.e. the number of atomic services within a service composition, impacts the computational performance of weak point recommendation algorithms, we perform experiments for 10 additional data sets. Each data set consists out

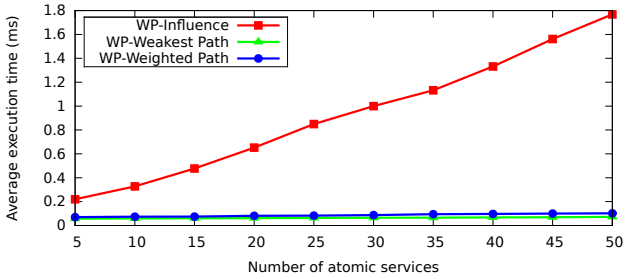


Fig. 12: Execution times.

of a 1000 applications having different number of atomic services. The number of services varies from 5 in the first data set to 50 services in the final data set with a step of 5 services. For each dataset we measure the average execution time to recommend a single list of weak points as follows:

$$AvgExec = \frac{\sum_{i=1}^{N} t_i}{N} \tag{18}$$

where t_i is the execution time for the i^{th} composition in the data set and N is the data set size. All the experiments were conducted on an Intel Core 2 Quad 2.66GHz CPU with 8GB of memory, running Ubuntu 11.10. The results are given by the graph in Fig. 12.

The results expectedly indicate that the average execution time for the *WP-Influence* algorithm is linearly dependent on the number of atomic services in a composition. This is due to the fact that the algorithm performs a sensitivity test for each atomic service separately requiring multiple reliability recalculations. On the other hand, heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath* calculate a list of weak points by traversing the model only once. This makes them less dependent on the number of atomic service nodes in the model as can be observed in Fig. 12. The observed increase in average execution time is due to the increase in size of the reliability models. In addition, since computations done by the *WP-WeightedPath* are heavier then in case of the *WP-WeakestPath*, the *WP-WeightedPath* algorithm expectedly shows higher average execution time than the *WP-WeakestPath*. In summary, the line showing average execution time for *WP-Influence* algorithm has a 105.3 and 48.4 times higher gradient then lines of *WP-WeakestPath* and *WP-WeightedPath* algorithms respectively.

Finally, we analyze the overall algorithm performance with regard to spent system resources by observing the average reliability growth rate. We define the average reliability growth rate measure as follows:

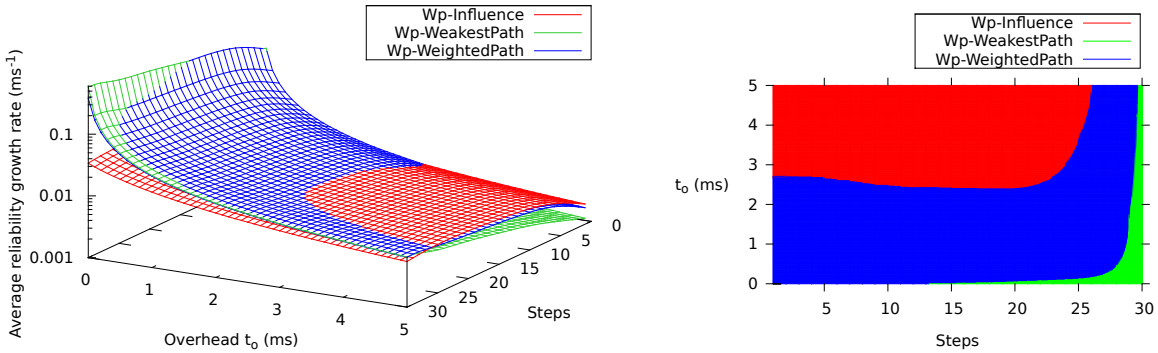
$$AvgGrRate_i = \frac{\sum_{j=1}^{N}(TotRelGr_{i,j}/t_{i,j})}{N} \tag{19}$$

where $TotRelGr_{i,j}$ is the total reliability growth for composition j after i improvement steps and $t_{i,j}$ total execution time from initial to the i^{th} reliability improvement step for composition j.

Up until now we have considered that no additional computational overhead per improvement step exists other then the weak point recommendation process. However, other overheads, like locating a replacement service or estimating the reliability of an atomic service, impact the overall performance of the proposed improvement method. For the purposes of the experiments we define all the additional overhead t_o as a fixed value for all the improvement steps performed on all compositions. Following this simplification we redefine the total execution time in (19) as $t'_{i,j} = t_{i,j} + it_o$, where i is the number of performed improvement steps.

In Fig. 13a we show the impact of the additional overhead t_o on the average reliability growth rate for a data set which consists out of 5000 composite applications built using 30 atomic services. The average growth rate, shown on the z-axis in logarithmic scale, is calculated for each of the 30 possible improvement steps, as denoted by the y-axis. The results indicate that for different values of t_o, different algorithms achieve highest reliability growth rate given the number of conducted improvement steps. To give a better overview which algorithm performs best for the given t_o and the conducted number of improvement steps, we show a projection of graph in Fig. 13a onto the x-y plane in the Fig. 13b. It can be concluded that for the presented experimental setup, the number of steps in which the *WP-Influence* and *WP-WeightedPath* algorithms achieve the highest *AvgGrRate* grows rapidly with the overhead t_o in a step-like manner followed by a region of more moderate growth. The presented results can be used to reason about the design of a hybrid algorithm which would achieve better performance by utilizing the best performing weak point recommendation algorithm.

We further support this claim by analyzing the experimental setup $t_o = 0ms$. The results shown in Fig. 14 indicate that both *WP-WeakestPath* and *WP-WeightedPath* algorithms achieve a higher average reliability growth rate then the *WP-Influence* for all improvement steps. Furthermore, the *WP-WeightedPath* has a greater initial average growth rate than the *WP-WeakestPath*, but lower average growth rate for later improvement steps. The presented results lead to a conclusion that better performance for the given data set could be achieved if initial improvement steps are done by utilizing *WP-WeightedPath* algorithm and the final steps by using *WP-WeakestPath* algorithm. Thus, the solution space can first be reduced using a more accurate but also more computationally demanding algorithm. Then a less accurate algorithm can be applied on the reduced solution space to achieve better average reliability growth rate. To support this claim, we present experimental results for a hybrid algorithm in which the first half of improvement steps (15 steps) are conducted by the *WP-WeightedPath* algorithm and the last half of improvement steps by the *WP-WeakestPath* algorithm. The results show that the hybrid algorithm achieves up to 25.77% better average reliability growth rate in the final improvement steps than the *WP-WeightedPath* algorithm. However,



(a) Average growth rate for t_o $0ms - 5ms$

(b) Best performing algorithm in relation to conducted improvement steps and overhead t_o

Fig. 13: Impact of the overhead t_o on the $AvgGrRate$

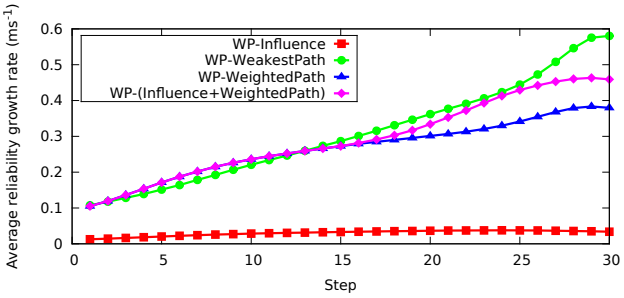


Fig. 14: Average reliability growth rate ($t_o = 0ms$).

the hybrid algorithm shows lower average reliability growth rate in the final improvement steps than the *WP-WeakestPath* algorithm. This is due to the fact that the *WP-WeightedPath* component of the hybrid algorithm has exhausted better service improvement solutions in earlier steps.

6 Related Work

Reliability of Atomic Services

A crucial prerequisite in estimating service composition reliability is the ability to reason about reliability of each atomic service incorporated into the workflow. To that end, there are two basic approaches that can be taken: *reliability estimation* and *prediction*. *Reliability estimation* is performed by applying formal reliability models that leverage past usage data that can be obtained through *service monitoring* approaches. Although a variety of such approaches have been suggested in the literature [29], [30], [31], [32] their application can be limited in practice, depending on the level of invasiveness. Specifically, accuracy of reliability estimation depends on the quantity and quality of the collected data whose attainment can in turn lead to degradations in performance of the monitored services. One way to mitigate this issue is by using *reliability prediction* methods. Traditionally, these methods can be conducted by fitting the collected usage data onto a reliability model in order to predict future performance. However, other promising prediction approaches that leverage statistical methods have emerged recently [33], [34]. In our previous research efforts we considered atomic service reliability prediction based on collaborative filtering [35] and k-means clustering approaches [36].

Service Selection

When considering what services are to be included into the composition, e.g. as part of an improvement solution, reliability is not the only applicable selection criterion. For instance, other dependability and security properties (e.g. maintainability, safety) [37], [38], and QoS parameters (e.g. response time, price, popularity) [15], [39], [40] can be taken into account. In that case, service selection is usually treated as an optimization problem where multiple parameters are aggregated into a single quality value. Specifically, it is necessary to compute the service selection that maximizes the aggregated quality value. However, such problems in general rely on computationally demanding solution strategies, like *NP-hard* integer programming. Promising approaches were presented by authors in [13] and [15] where a combination of local and global optimization methods was used to achieve better computational performance at a low cost of accuracy. Our reliability improvement method can be used to make the presented approaches more computationally efficient by reducing the solution space, i.e. by focusing the optimization problem on most relevant services. In addition, atomic services' influence on the overall reliability can be used as an additional optimization parameter. Alternatively, our method can be extended to take into account additional parameters. If a probability distribution function and thresholds for acceptable parameter values are know, the aggregated quality value can be represented as a random variable. In that case, the presented method can be applied as described in this paper, taking into account aggregated quality value, rather than reliability.

Dynamic Service Composition

Issues regarding reliability of service compositions are dominantly handled at run time. This is due to the fact that service *QoS* parameters can change unpredictably

and, thus, run time dynamic service selection is an appropriate method to make service compositions more robust to failures. However, reconfiguring service compositions at run time introduces additional issues, like statefullness and composability [7]. For that reason, reliability models used to perform dynamic service selections are dominantly state-based. One of the principal disadvantages of state-based models is their proneness to state explosion for more complex composite workflows. A promising approach to handle state explosions, based on the bounded set technique, was given in [14]. However such an approach can be impractical when performing extensive service monitoring leads to significant decrease in performance. By applying the method proposed in our paper, monitoring and state related issues could be reduced by focusing on a subset of atomic services.

Reliability Sensitivity Analysis

The reliability model proposed in this paper is based on observing the inner structure of a service composition. Similar architecture-based approaches have been considered in previous research efforts. In [41] the authors suggest building a structure analysis chart based on the BPEL representation. Similarly, in [24] the authors use the BPEL structure to construct a flow graph model and present a recursive algorithm used to compute the reliability of a service composition. Furthermore, Zhong et. al. propose in [42] an approach to convert the BPEL structure into a stochastic Petri network and give solution how to calculate the overall service composition reliability. On the other hand, in [43] the authors propose a discrete time Markov chain model based on modeling and aggregating all the possible service composition execution scenarios. All the stated approaches can be leveraged to perform sensitivity analysis with aim of determining how do individual atomic services impact the overall reliability. However, the authors do not discuss how to perform the analysis in a computationally efficient way that would be appropriate for large-scale service compositions.

7 Conclusions and Future Work

This paper presents a novel reliability improvement method for SOA-based applications that aims to address the shortcomings in development of large component based systems. The method leverages a service composition reliability model based on belief networks to recommend the most reliability-critical workflow components, i.e. weak points. To recommend the weak points in a scalable way, a suite of recommendation algorithms with varying complexity and accuracy is applied. The algorithms were evaluated on data sets of randomly generated service composition reliability models. The results confirm that *WP-Influence* is the most accurate algorithm, as when compared to the optimal solution it achieves several orders of magnitude lower mean absolute percentage error (*MAPE*) than the heuristic algorithms. Furthermore, the results conclusively indicate that *WP-WeightedPath* is more accurate than *WP-WeakestPath* as it achieves up to 3.26 times lower *MAPE*. On the other hand, the results for average execution time indicate that *WP-Influence* is most computationally demanding, while *WP-WeightedPath* is more computationally demanding than *WP-WeakestPath*. Computational performance of both heuristic algorithms is much less dependent on the reliability model size than in case of *WP-Influence*. Finally, impact of additional computational overhead per improvement step is evaluated. The results suggest that if step-wise overhead is high, it is more efficient to apply the *WP-Influence* algorithm as it, due to higher accuracy, requires less improvement steps. On the other hand, heuristic algorithms prove to be more efficient when the additional overhead is low, as well as in the final improvement steps when the solution space is reduced.

One of our future research directions will be to further explore heuristics for the weak point recommendation algorithms. Furthermore, our goal is to evaluate the proposed reliability improvement method on a statistically relevant set of real-world compositions. To that end, a prospect research direction includes application of the proposed method to data sets gathered form mashup tools, like *Yahoo! Pipes* [44] and *Geppeto* [45].

Acknowledgments

The authors acknowledge the support of the Ministry of Science, Education, and Sports of the Republic of Croatia through the Computing Environments for Ubiquitous Distributed Systems (036-0362980-1921) research project.

References

[1] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[2] W3C, "Web Services Architecture," http://www.w3.org/TR/ws-arch/, February 2004.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Berlin: Springer, 2004.

[4] D. Ma, "The business model of "software-as-a-service"," in *Services Computing, 2007. SCC 2007. IEEE International Conference on*, july 2007, pp. 701 –702.

[5] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Dec. 2008.

[6] Q. Liang, X. Wu, and H. C. Lau, "Optimizing service systems based on application-level qos," *Services Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 108 –121, april-june 2009.

[7] S.-Y. Hwang, E.-P. Lim, C.-H. Lee, and C.-H. Chen, "Dynamic web service selection for reliable web service composition," *Services Computing, IEEE Transactions on*, vol. 1, no. 2, pp. 104 –116, april-june 2008.

[8] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 311 – 327, may 2004.

[9] M. Aime, A. Lioy, and P. Pomi, "Automatic (re)configuration of it systems for dependability," *Services Computing, IEEE Transactions on*, vol. 4, no. 2, pp. 110 –124, april-june 2011.

[10] K. Fujii and T. Suda, "Semantics-based context-aware dynamic service composition," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 12:1–12:31, May 2009.

[11] H. Hanen and J. Bourcier, "Dependability-Driven Runtime Management of Service Oriented Architectures," in *PESOS - 4th International Workshop on Principles of Engineering Service-Oriented Systems - 2012*, Zurich, Suisse, Jun. 2012.
[12] S. Kalasapur, M. Kumar, and B. Shirazi, "Dynamic service composition in pervasive computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 7, pp. 907 –918, july 2007.
[13] M. Alrifai, T. Risse, and W. Nejdl, "A hybrid approach for efficient web service composition with end-to-end qos constraints," *ACM Trans. Web*, vol. 6, no. 2, pp. 7:1–7:31, Jun. 2012.
[14] H. Mansour and T. Dillon, "Dependability and rollback recovery for composite web services," *Services Computing, IEEE Transactions on*, vol. 4, no. 4, pp. 328 –339, oct.-dec. 2011.
[15] Z. Zheng and M. Lyu, "A qos-aware fault tolerant middleware for dependable service composition," in *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, 29 2009-july 2 2009, pp. 239 –248.
[16] W. Liu and W. Wong, "Discovering homogenous service communities through web service clustering," in *Proceedings of the 2008 AAMAS international conference on Service-oriented computing: agents, semantics, and engineering*, ser. SOCASE'08, 2008, pp. 69–82.
[17] X. Liu, G. Huang, and H. Mei, "Discovering homogeneous web service community in the user-centric web environment," *IEEE Transactions on Services Computing*, vol. 2, pp. 167–181, 2009.
[18] X. Zhang, Y. Yin, M. Zhang, and B. Zhang, "Web service community discovery based on spectrum clustering," in *Computational Intelligence and Security, 2009. CIS '09. International Conference on*, vol. 2, dec. 2009, pp. 187 –191.
[19] C. Perryea and S. Chung, "Community-based service discovery," in *Web Services, 2006. ICWS '06. International Conference on*, sept. 2006, pp. 903 –906.
[20] G. Delac, M. Silic, and S. Srbljic, "Reliability modeling for soa systems," in *MIPRO, 2012 Proceedings of the 35th International Convention*, may 2012, pp. 847 –852.
[21] OASIS, "Web services business process execution language version 2.0," http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007, OASIS Standard.
[22] W. van der Aalst and A. ter Hofstede, "Yawl: yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245 – 275, 2005.
[23] S. Srbljic, D. Skvorc, and D. Skrobo, "Programming language design for event-driven service composition." *AUTOMATIKA Journal for Control, Measurement, Electronics, Computing and Communications*, vol. 51, no. 4, pp. 374–386, 2011.
[24] V. Cortellessa and V. Grassi, "Reliability modeling and analysis of service-oriented architectures," in *Test and Analysis of Web Services*. Springer Berlin Heidelberg, 2007, pp. 339–362.
[25] M. Winkler, T. Springer, E. Trigos, and A. Schill, "Analysing dependencies in service compositions," in *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, ser. Lecture Notes in Computer Science, 2010, vol. 6275, pp. 123–133.
[26] B. Randell and J. Xu, "The evolution of the recovery block concept," in *IN SOFTWARE FAULT TOLERANCE*. John Wiley & Sons Ltd, 1994, pp. 1–22.
[27] A. Avizienis, "The n-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 12, pp. 1491 – 1501, dec. 1985.
[28] D. F. McAllister and M. A. Vouk, "Fault-tolerant software reliability engineering," in *Handbook of software reliability engineering*, M. R. Lyu, Ed., 1996, pp. 567–614.
[29] Z. Zheng and M. Lyu, "Ws-dream: A distributed reliability assessment mechanism for web services," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, june 2008, pp. 392 –397.
[30] C. Ghezzi and S. Guinea, "Run-time monitoring in service-oriented architectures," in *Test and Analysis of Web Services*. Springer Berlin Heidelberg, 2007, pp. 237–264.
[31] C. A. Ardagna, R. Jhawar, and V. Piuri, "Dependability certification of services: a model-based approach," *Computing*, pp. 1–28, 2013.
[32] R. C. Cheung, "A user-oriented software reliability model," *IEEE Trans. Softw. Eng.*, vol. 6, no. 2, pp. 118–125, Mar. 1980.
[33] Z. Zheng, H. Ma, M. Lyu, and I. King, "Qos-aware web service recommendation by collaborative filtering," *Services Computing, IEEE Transactions on*, vol. 4, no. 2, pp. 140 –152, april-june 2011.
[34] Z. Zheng and M. Lyu, "Collaborative reliability prediction of service-oriented systems," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, may 2010, pp. 35 –44.
[35] M. Silic, G. Delac, I. Krka, and S. Srbljic, "Scalable and accurate prediction of availability of atomic web services," *Services Computing, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013.
[36] M. Silic, G. Delac, and S. Srbljic, "Prediction of atomic web services reliability based on k-means clustering," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 70–80.
[37] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.
[38] M. Anisetti, C. Ardagna, E. Damiani, and J. Maggesi, "Security certification-aware service discovery and selection," in *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, Dec 2012, pp. 1–8.
[39] S. Ran, "A model for web services discovery with qos," *SIGecom Exch.*, vol. 4, no. 1, pp. 1–10, Mar. 2003.
[40] Y. Hao, Y. Zhang, and J. Cao, "A novel qos model and computation framework in web service selection," *World Wide Web*, vol. 15, no. 5-6, pp. 663–684, 2012.
[41] B. Li, X. Fan, Y. Zhou, and Z. Su, "Evaluating the reliability of web services based on bpel code structure analysis and run-time information capture," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 30 2010-dec. 3 2010, pp. 206 –215.
[42] D. Zhong and Z. Qi, "A petri net based approach for reliability prediction of web services," in *Proceedings of the 2006 international conference on On the Move to Meaningful Internet Systems*, ser. OTM'06, 2006, pp. 116–125.
[43] R. Yingxin, G. Qing, Q. Jingxian, and C. Daoxu, "Reliability prediction of web service composition based on dtmc," in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, july 2009, pp. 369 –375.
[44] M. Pruett, *Yahoo! pipes*, 1st ed. O'Reilly, 2007.
[45] S. Srbljic, D. Skvorc, and D. Skrobo, "Widget-oriented consumer programming," *AUTOMATIKA Journal for Control, Measurement, Electronics, Computing and Communications*, vol. 50, no. 3–4, pp. 252–264, 2009.

Goran Delac is a research and teaching assistant at the University of Zagreb, Faculty of Electrical Engineering and Computing. He received his Ph.D. in Computer Science from the University of Zagreb, Faculty of Electrical Engineering and Computing in 2014. His research interests span distributed systems, parallel computing, fault tolerant systems and service-oriented computing. He is a graduate student member of the IEEE.

Marin Silic is a research and teaching assistant at the University of Zagreb, Faculty of Electrical Engineering and Computing. He has a Ph.D. in Computer Science from the University of Zagreb Faculty of Electrical Engineering and Computing. His research interests span distributed systems, service-oriented computing, software engineering, software reliability. He is a graduate student member of the IEEE.

Sinisa Srbljic is a full professor at the University of Zagreb, Faculty of Electrical Engineering and Computing and head of Consumer Computing Lab. He was a visiting scientist at the Huawei, Santa Clara, USA, the GlobalLogic, San Jose, USA, the UC Irvine, USA, the AT&T Labs, San Jose, USA, and the University of Toronto, Canada. He is a computing visionary with unique experience in consumerization of innovation. He is a senior member of the IEEE.