Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems

Harald Psaier, Lukasz Juszczyk, Florian Skopik, Daniel Schall, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8, 1040 Wien, Austria
{lastname}@infosys.tuwien.ac.at

***Abstract*—Mixed service-oriented systems composed of human actors and software services build up complex interaction networks. Without any coordination, such systems may exhibit undesirable properties due to unexpected behavior. Also, communications and interactions in such networks are not preplanned by top-down composition models. Consequently, the management of service-oriented applications is difficult due to changing interaction and behavior patterns that possibly contradict and result in faults from varying conditions and misbehavior in the network. In this paper we present a self-adaptation approach that regulates local interactions to maintain desired system functionality. To prevent degraded or stalled systems, adaptations operate by link modification or substitution of actors based on similarity and trust metrics. Unlike a security perspective on trust, we focus on the notion of socially inspired trust. We design an architecture based on two separate independent frameworks. One providing a real Web service testbed extensible for dynamic adaptation actions. The other is our self-adaptation framework including all modules required by systems with self-* properties. In our experiments we study a trust and similarity based adaptation approach by simulating dynamic interactions in the real Web services testbed.**

***Index Terms*—Service-oriented collaboration, monitoring, self-adaptation, web service testbed, dynamic trust**

I. Introduction

Service-oriented architectures (SOA) implementations are typically designed as large-scale systems. Applications are composed from the capabilities of distributed services that are discovered at runtime. Dynamic loosely bound systems make the management of large-scale distributed applications increasingly complex. Adaptations are necessary to keep the system within well-defined boundaries such as expected load or desired behavior. Changing requirements and flexible utilization demand for comprehensive analysis of the resulting effects prior to integration. Changes interfere with established services, connections, or policies and on top of all affect dependencies. However, service compositions must be maintained and adapted depending on predefined runtime properties such as quality of service (QoS) [1] and behavior [2].

In this work we propose a monitoring and self-adaptation approach of *service-oriented collaboration networks*. We consider systems that are based on the capabilities of human actors, defined as Human-Provided Services (HPSs) [3] and traditional Software-Based Services (SBSs). The integration of humans and software-based services is motivated by the difficulties to adopt human expertise into software implementations. Instead of dispensing with human capabilities, people handle tasks behind traditional service interfaces. In contrast to process-centric flows (top-down compositions), we advocate flexible compositions wherein services can be added at any time exhibiting new behavior properties. However, especially the involvement of and dependencies on humans as a part of flexible compositions makes the functioning of applications difficult to determine. Heterogeneity has a major impact on all aspects of the system since system dynamics and evolution are driven by software services and human behavior [2]. A main challenge is to monitor, analyze, and evaluate specific behaviors which may affect system performance or reliability.

We present a solution to this problem based on an architecture including a Web services testbed [4] at its core. The testbed allows to simulate and track the effects on a composition resulting from different environmental conditions. The success of self-adaptation strategies commonly depends on the recognition of the system's current state and potential actions to achieve desired improvements.

This paper presents the following novel key contributions:

- Modeling and simulating human behavior in service-oriented collaboration networks.
- A flexible interaction model for service-oriented systems. The interaction model is based on delegation actions performed by actors. Associated tasks are routed through the system following standard WS-Addressing techniques.
- Models for misbehavior and related repair actions to prevent inefficient or degraded system performance. We identify *delegation factory* and *delegation sink* and their behavior.
- Discovery of delegation receivers to prevent or mitigate misbehavior. We present a novel trust metric based on profile similarity measurements.

The paper's structure is as follows. Section II provides a motivating scenario for service-oriented collaboration systems. Section III explains the concepts of similarity and trust used for adaptation strategies. Section IV outlines the twofold system architecture. Section V details the aspects of behavior monitoring. Experiments and results are discussed in Section VI followed by related work in Section VII. Section VIII concludes the paper.

II. On Self-adaptation in Collaborative SOA

The goal of self-adaptation in service-oriented systems is to prevent the running system from the trend to an unexpected low performance. As in autonomic computing the aim is to create robust dependable self-managing systems [5]. The established methodology [6], and the one of self-adaptive systems [7] is to design and implement a *control-feedback loop*. This feedback loop is known as the *MAPE* cycle consisting of four essential steps: *monitor, analyze, plan*, and *execute*. Systems that adapt themselves autonomously are enhanced with *sensors and effectors* that allow *network model* creation and *adaptation strategies*. This provides the necessary self-awareness to manage the system autonomously.

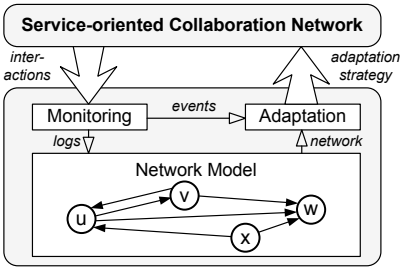


Fig. 1. Self-adaptation and behavior monitoring approach.

Figure 1 illustrates the proposed approach to manage and adapt service-oriented collaboration networks. Such systems comprise different kinds of actors, services, and compositions thereof. Interactions are captured from the system through interceptor and logging capabilities. The *monitoring* component feeds interaction logs into a network representation of actors and their relations. Behavior patterns are analyzed based on a *network model*. A *self-adaptation* engine evaluates policies to trigger potential adaptation strategies. Adaptations include structural change (link modification) and actor substitution.

Our approach with two frameworks allows testing of adaptation strategies in versatile service-based application scenarios. Examples are *crowdsourcing* applications [8] in enterprise environments or open Internet based platforms. These online platforms distribute problem-solving tasks among a group of humans. Crowdsourcing follows the 'open world' assumption allowing humans to provide their capabilities to the platform by registering themselves as services. Some of the major challenges [9] are monitoring of crowd capabilities, detection of missing capabilities, strategies to gather those capabilities, and tasks' status tracking. In the following we discuss collaborations in service-oriented networks.

Processes in collaborative environments are not restricted to single companies only, but may span multiple organizations, sites, and partners. External consultants and third-party experts may be involved in certain steps of such processes. These actors perform assigned tasks with respect to prior negotiated agreements. Single task owners may consume services from external expert communities. A typical use case is the evaluation of experiment results and preparation of test reports in biology, physics, or computer science by third-party consultants (i.e., the network of experts). While the results of certain simple but often repeated experiments can be efficiently processed by software services, analyzing more complex data usually needs human assistance. We model a mixed expert network consisting of Human-Provided and Software-Based Services belonging to different communities. The members of these communities are discovered based on their main expertise areas, and connected through certain relations (detailed in the following sections). Community members receive requests from external service consumers, process them, and respond to the requests. Our environment uses standardized SOA infrastructures, relying on widely adopted standards, such as SOAP and the Web Service Description Language (WSDL), to combine the capabilities of humans and software services.

Various circumstances may cause inefficient task assignments in expert communities. Performance degradations can be expected when a minority of distinguished experts become flooded with tasks while the majority remains idle. Load distribution problems can be compensated with *delegations* [10], [11]. Each expert in a community is connected to other experts that may potentially receive delegations. We assume that experts delegate work they are not able to perform because of missing mandatory skills or due to overload conditions. Delegation receivers can accept or reject task delegations. Community members usually have explicit incentives to accept tasks, such as collecting rewards for successfully performed work to increase their community standing (reputation). Delegations work well as long as there is some agreement on members' *delegation behavior*: How many tasks should be delegated to the same partner in a certain time frame? How many task can a community member accept without neglecting other work? However, if misbehavior cannot be avoided in the network, its effects need to be compensated. We identify two types of misbehavior: *delegation factory* and *delegation sink*.

A **delegation factory** produces unusual (i.e., unhealthy) amounts of task delegations, leading to a performance degradation of the entire network. For example (see Figure 1), if a node v accepts large amounts of tasks without actually performing them, but simply delegates to one of its neighboring nodes (e.g., w). Hence, v's misbehavior produces high load at the neighboring node w. Work overloads lead to delays and, since tasks are blocked for a longer while, to a performance degradation from a global network point of view. A **delegation sink** can be characterized by the following behavior. Node w accepts more task delegations from u, v, and x as it is actually able to handle. In our collaborative network, this may happen due to the fact that w either underestimates the workload or wants to increase its reputation as a valuable collaboration partner in a doubtful manner. Since w is actually neither able to perform all tasks nor to delegate to colleagues (because of missing outgoing delegation links), accepted tasks remain in its task pool. Again, we observe misbehavior as the delegation receiver causes blocked tasks and performance degradation from a network perspective.

Our approach provides a testing environment for such applications to address related challenges.

III. Profile Similarity and Dynamic Trust

Collaborative networks, as outlined in the previous sections, are subject to our trust studies. Unlike a security view, we focus on the notion of dynamic trust from a social perspective [12]. We argue that trust between community members is essential for successful collaborations. The notion of dynamic trust refers to the interpretation of previous collaboration behavior [10], [13] and considers the similarity of dynamically adapting skills and interests [14], [15].

Especially in collaborative environments, where users are exposed to higher risks than in common social network scenarios, and where business is at stake, considering trust is essential to effectively guide human interactions. In this paper, we particularly focus on the establishment of trust through measuring interest similarities [10]:

- *Trust Mirroring* implies that actors with similar profiles (interests, skills, community membership) tend to trust each other more than completely unknown actors.
- *Trust Teleportation* rests on the similarity of human or service capabilities, and describes that trust in a member of a certain community can be teleported to other members. For instance, if an actor, belonging to a certain expert group, is trusted because of his distinguished knowledge, other members of the same group may benefit from this trust relation as well.

A. Interest Profile Creation

In contrast to common top-down approaches that apply taxonomies and ontologies to define certain skills and expertise areas, we follow a mining approach that addresses inherent dynamics of flexible collaboration environments. In particular, skills and expertise as well as interests change over time, but are rarely updated if they are managed manually in a registry. Hence, we determine and update them automatically through mining.

The creation of interest profiles without explicit user input has been studied in [10]. As discussed before, interactions, i.e., delegation requests, are tagged with keywords. As delegation receivers process tasks, our system is able to learn how well people cope with certain tagged tasks; and therefore, able to determine their centers of interests. We use task keywords to create dynamically adapting interest profiles based on tags and manage them in a vector space model.

The utilized concepts are well-known from the area of information retrieval (see for instance [16]). However, while they are used to determine the similarities of given documents, we create these documents (that reflect user profiles) from used tags dynamically on the fly.

The profile vector $\mathbf{p_u}$ of actor u in Eq. (1) describes the frequencies f the tags $T = \{t_1, t_2, t_3 \dots\}$ are used in delegated tasks accepted by actor u.

$$\mathbf{p_u} = \langle f(t_1), f(t_2), f(t_3) \dots \rangle \tag{1}$$

The tag frequency matrix $\mathfrak{T}$ (2) in Eq. 2, built from profile vectors, describes the frequencies of used tags $T = \{t_1, t_2, t_3 \dots\}$ by all actors $A = \{u, v, w \dots\}$.

$$\mathfrak{T} = \langle \mathbf{p_u}, \mathbf{p_v}, \mathbf{p_w} \dots \rangle_{|T| \times |A|} \tag{2}$$

The popular tf^*idf model [16] introduces tag weighting based on the relative distinctiveness of tags; see Eq. (3). Each entry in $\mathfrak{T}$ is weighted by the log of the total number of actors $|A|$, divided by the amount $n_t = |\{u \in A \mid tf(t,u) > 0\}|$ of actors who used tag t.

$$tf^*idf(t,u) = tf(t,u) \cdot \log \frac{|A|}{n_t} \tag{3}$$

Finally, the cosine similarity, a popular measure to determine the similarity of two vectors in a vector space model, is applied to determine the similarity of two actor profiles $\mathbf{p_u}$ and $\mathbf{p_v}$; see Eq. (4).

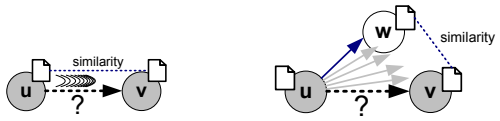
$$sim_{profile}(\mathbf{p_u}, \mathbf{p_v}) = \cos(\mathbf{p_u}, \mathbf{p_v}) = \frac{\mathbf{p_u} \cdot \mathbf{p_v}}{||\mathbf{p_u}|| \; ||\mathbf{p_v}||} \tag{4}$$

B. The Interplay of Interest Similarity and Trust

In our model, a trust relation $\tau(u, v)$ mainly relies on the interest and expertise similarities of actors. We apply various concepts to facilitate the emergence of trust among network members.

Trust Mirroring. Trust τ_{mir} (Figure 2(a)) is typically applied in environments where actors have the same roles (e.g., online social platforms). Depending on the environment, interest and competency similarities of people can be interpreted directly as an indicator for future trust (Eq. 5). There is strong evidence that actors 'similar minded' tend to trust each other more than any random actors [12], [15]; e.g., movie recommendations of people with same interests are usually more trustworthy than the opinions of unknown persons. Mirrored trust relations are directed, iff $sim_{profile}(\mathbf{p_u}, \mathbf{p_v}) \neq sim_{profile}(\mathbf{p_u}, \mathbf{p_v})$. For instance an experienced actor v might have at least the same competencies as a novice u. Therefore, v covers mostly all competencies of u and $\tau_{mir}(u, v)$ is high, while this is not true for $\tau_{mir}(v, u)$.

$$\tau_{mir}(u, v) = sim_{profile}(\mathbf{p_u}, \mathbf{p_v}) \tag{5}$$



(a) Trust Mirroring. (b) Trust Teleportation

Fig. 2. Concepts for the establishment of trust through interest similarities.

Trust Teleportation. Trust τ_{tele} is applied as depicted by Figure 2(b). We assume that u has established a trust relationship to w in the past, for example, based on w's capabilities to assist u in work activities. Therefore, others having interests and capabilities similar to w may become similarly trusted by u in the future. In contrast to mirroring, trust teleportation may also be applied in environments comprising actors with

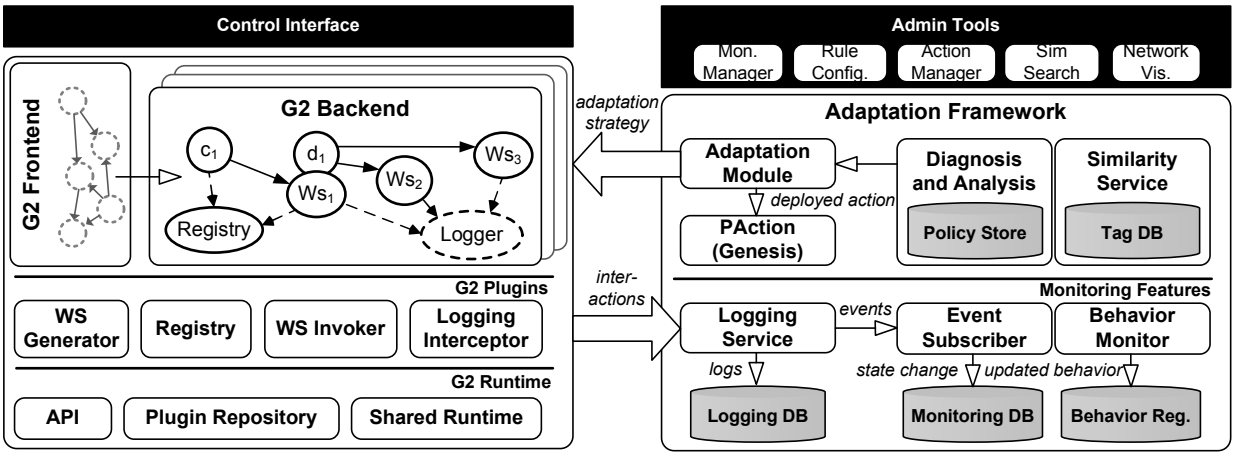


Fig. 3. Architecture for self-adaptation in service-oriented systems.

different roles. For example, a manager might trust a software developer belonging to a certain group. Other members in the same group may benefit from the existing trust relationship by being recommended as trustworthy as well. We attempt to predict the amount of future trust from u to v by comparing w's and v's profiles P.

$$\tau_{tele}(u,v) = \frac{\sum_{w \in M'} \tau(u,w) \cdot (sim_{profile}(\mathbf{p_w}, \mathbf{p_v}))^2}{\sum_{w \in M'} sim_{profile}(\mathbf{p_w}, \mathbf{p_v})} \quad (6)$$

Equation 6 deals with a generalized case where several trust relations from u to members of a group M' are teleported to a still untrusted actor v. Teleported relations are weighted and attenuated by the similarity measurement results of actor profiles.

IV. Design and Architecture

This section provides an overview of the components and services that allow simulatons and tests of adaptation scenarios in collaborative service-oriented systems. Our architecture (see Figure 3) consists of two main building blocks: the *testbed runtime environment* based on the Genesis2 framework [4] and the VieCure *adaptation* and *self-healing framework*, partly adopted from our previous work [2]. The integration of both systems enables the realization of the *control-feedback loop* as illustrated in Figure 1.

A. Genesis2 Testbed Generator Framework

The purpose of the Genesis2 framework (in short, G2) is to support software engineers in setting up testbeds for runtime evaluation of SOA-based concepts and implementations; in particular also collaboration networks. It allows to establish environments consisting of services, clients, registries, and other SOA components, to program the structure and behavior of the whole testbed, and to steer the execution of test cases on-the-fly. G2's most distinct feature is its ability to generate real testbed instances (instead of just performing simulations) which allows engineers to integrate these testbeds into existing SOA environments and, based on these infrastructures, to perform realistic tests at runtime.

As depicted in Figure 3, the G2 framework comprises a centralized front-end, from where testbeds are modeled and controlled, and a distributed back-end at which the models are transformed into real testbed instances. The front-end maintains a virtual view on the testbed, allows engineers to manipulate it via Groovy [17] scripts, and propagates changes to the back-end in order to adapt the running testbed. To ensure extensibility, G2 follows a modular approach where a base runtime framework provides a functional grounding for composable plugins. These augment the testbed's functionality, making it possible to emulate diverse topologies, functional and non-functional properties, and behavior. Furthermore, each plugin registers itself at the shared runtime in order offer its functionality via the framework's script API.

The sample script in Listing 1 demonstrates a specification of a Web service which queries a registry plugin, applies a delegation strategy, and forwards the request message to a worker service. First, a call interceptor is created and customized with a Groovy closure which passes the SOAP message to the logger plugin. Then, a data type definition is imported from an XML Schema file for being later applied as a message type for the subsequently defined web service `Proxy`. The proxy service first attaches the created call interceptor to itself and defines an operation which delegates the request. This procedure is split into querying the registry for tagged Web services, applying the delegation strategy (`dStrat`) for determining the destination, and invoking the `Process` operation on it. For later adaptations, the delegation behavior itself is not hardcoded into the operation but outsourced as a service variable containing the delegation code. This makes it possible to update the deployed service's behavior at runtime by replacing the variable. Finally, in Lines 24 and 25 a back-end host is referenced and the proxy service is deployed on it. Due to space constraints, this demo script does only cover a heavily restricted specification of the testbed and also lacks the definition of other participants, such as worker services

```
li=callinterceptor.create() // logging interceptor
li.hooks=[in:"RECEIVE", out :"PRE_STREAM"] // bind to phases
li.code={ctx −> logger.logToDB(ctx.soapMsg) } // process msg

msgType=datatype.create("file.xsd","typeName") // xsd import

sList=webservice.build {
  // create web service
  Proxy(binding:"doc,lit", namespace="http://...") {
    // attach logging interceptor
    interceptors+=li
    // create web service operation
    Delegate(input:msgType, resonse:msgType) {
      refs = registry.get{s−> "Worker" in s.tags} // by tag
      r = dStrat(refs)
      return r.Process(input).response
    }
    // delegation strategy as closure variable
    dStrat={ refs −> return refs[0] } // default: take first
  }
}

srv=sList[0] // only one service declared, take it
h=host.create("somehost:8181") // import back−end host
srv.deployAt(h) // deploy service at remote back−end host

srv.dStrat={ refs−> /*...*/ } // adapt strategy at runtime
```

Listing 1. Groovy script specifying delegator service.

```
<soap:Envelope
 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 xmlns:xsi="www.w3.org/2001/XMLSchema-instance"
 xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
 xmlns:hpsht="http://myhps.org/HumanTask"
 xmlns:vietypes="http://viete.infosys.tuwien.ac.at/Type"
 <soap:Header>
  <wsa:MessageID>052c5e11−5abd−4763−8725−86eaa48fb0fc</wsa:MessageID>
  <wsa:ReplyTo>http://www.expertnetwork.org/Actor#Harald</wsa:ReplyTo>
  <wsa:From>http://www.expertnetwork.org/Actor#Harald</wsa:From>
  <wsa:To>http://www.expertnetwork.org/Actor#Florian</wsa:To>
  <wsa:Action>http://myhps.org/Action/Delegation</wsa:Action>
  <vietypes:activity url="http://www.expertnetwork.org/Activity#42"/>
  <vietypes:timestamp value="2010-05-06T15:13:21"/>
  <hpsht:taskContext>
    <hpsht:deadline="2010-05-07T12:00:00"/>
    <hpsht:priority>
     <!−− task priority −−>
    </hpsht:priority>
    <hpsht:keywords>WS, Adaptation, Trust</hpsht:keywords>
  </hpsht:taskContext>
 </soap:Header>
 <soap:Body>
   <hps:prepareReport>
    <!−− details omitted −−>
   </hps:prepareReport>
 </soap:Body>
</soap:Envelope>
```

Listing 2. Simplified SOAP interaction example.

and clients for bootstrapping the testbed's activity. In our evaluation, we have applied G2 in order to have a customizable Web service testbed for verifying the quality of our concepts in realistic scenarios, e.g., for a detailed analysis of performance and scalability. For a more detailed description of the G2 framework and its capabilities we refer readers to [4].

B. Adaptation Framework

The adaptation framework is located on the right side in Figure 3. The framework has monitoring features including logging, eventing, and a component for capturing actor behavior. Based on observations obtained from the testbed, adaptation actions are taken.

- The *Logging Service* is used by the logger plugin (see `PLogger` in Figure 3). Logged messages are persistently saved in a database for analysis. The logging service also implements a publish/subscribe mechanism to offer distributed event notification capabilities. Subscribers can specify filters using XPath statements which are evaluated against received logged messages.
  A short example is shown in Listing 2. Header extensions (Line 7 - 22) include the context of interactions (i.e., the activity that is performed), delegation restrictions, identify the sender and receivers using WS-Addressing [18] mechanisms, and hold some meta-information about the activity type itself. `MessageIDs` enable message correlation to correctly match requests and responses. `Timestamps` capture the actual creation of the message and are used for message ordering. For HPSs, SOAP messages are mapped to user interfaces by the HPS framework [3]. `Task Context` related information is also transported via header mechanisms. While activities depict what kind of information is exchanged between actors (type system) and how collaborations are structured, tasks control the status of interactions and constraints in processing certain activities.
  Multiple instances of the logging service can be deployed to achieve scalability in large scale environments.
- *Event Subscribers* receive events based on filters that can be specified for different types of (inter-)actions, for example, to capture only delegation flows. Subscribers are used to capture the runtime state of nodes within the testbed environment such as current load of a node.
- The *Behavior Monitor* updates and stores periodically the actual interaction behavior of nodes as profiles in the *behavior registry*. This mechanism assists the following diagnosis to correlate environment events and behavior changes.
- *Diagnosis and Analysis* algorithms are initiated to evaluate the root cause of undesirable system states. Preconfigured triggers for such events, e.g., events reporting violations, inform the diagnosis module about deviations from desired behavior. Captured and filtered interaction logs as well as actual node behaviors assist in recognizing the system's health state.
- The *Similarity Service* uses the tag database to search for actors based on profile keywords (i.e., to replace an actor or to establish a new link to actors). Tags are obtained from logged interactions.
- The *Adaptation Module* deployed appropriate adaptation actions. An example for an adaptation action is to update a node's *delegation strategy* as indicated in Figure 3. For that purpose, the *PAction* plugin communicates with G2's control interface.

A set of Web-based *Admin Tools* have been implemented to offer graphical user interfaces for configuring and visualizing the properties of testbeds. User tools include, for example, policy design for adaptations or visualizations of monitored interactions.

V. Behavior Monitoring and Self-Adaptation

The design of the architecture presented in the previous section provides a variety of possibilities for self-adaptation strategies. Figure 3 shows that the adaptation framework is loosely coupled to the testbed. Furthermore, logging interactions is a very generic approach to monitor the environment. The focus of this paper is adaptation of service misbehavior. Misbehavior appears on any unexpected change of behavior of a testbed component with noticeable function degradation impacts to the whole or major parts of the testbed. Our monitoring and adaptation strategies follow the principle of smooth integration with least interference. However, a loosely coupled design often results in delayed and unclear state information. This can cause a possibly delayed deployment and application of adaptations. On the other hand, the testbed remains more authentic and true to current real environments which lack direct monitoring and adaptation functionality.

Monitoring in this architecture relies on the accuracy and timeliness of the *Logging Service*. *Diagnosis and Analysis* get all required status updates with the help of the *Event Subscriber* mechanism. Filtered status information populates the network model held by *Diagnosis and Analysis* module. During start-up the first interaction information is used to build the initial structure of the model. During runtime this information synchronizes the model with actual status changes observed on the network. Especially the interaction data filtered by the *Behavior Monitor* module allows *Diagnosis and Analysis* to draw conclusions from interactions about possible misbehavior at the services.

Detectable misbehavior patterns are described in the *Policy Store* together with related recovery strategies. The components of the store include trigger, diagnosis and recovery action modules (cf., [2]). Whilst the trigger defines potential misbehavior in a rule, the fired diagnosis analyzes the detected incident using its network model. The model information in combination with current interaction facts from the log history is used to estimate the necessary recovery actions. Finally, recovery strategies are estimated and deployed to adapt the real network. Referring, e.g., to the misbehavior patterns presented in Section II a sink behavior trigger could be expressed according to the previously given description by a threshold value defining an admissible amount of tasks at a monitored node. A fired diagnosis would further inspect the delegation history of a suspected node by consulting its task delegation log data an integral part of its network model. If a sink behavior is identified the diagnosis plans recovery actions. Actions are situation dependent and there are possibly multiple options for recovery.

In this paper the recovery approach is to reconfigure the network by adapting the interaction channels between the service nodes. Channels are opened to provide new interactions to alternative nodes and closed to hinder misbehaving nodes to further affect the surrounding nodes and degrade the environment's function. The challenge is not only to detect misbehaving nodes but also to find alternative interaction channels for those problem nodes. A feasible adaptation must temporarily decouple misbehaving nodes from the network and instantly find possible candidates for substitution. Potential candidates must expose similar properties as the misbehaving node, e.g., have similar capabilities, and additionally, have the least tendency to misbehavior, e.g., those with least current task load. In a real mixed system environment nodes' capabilities will change and the initial registered profiles will diverge with time from the current. Therefore our framework includes a *Similarity Service* that keeps track of the profile changes and provides alternatives to nodes according to their current snapshot profile.

In the following we show how the misbehavior patterns introduced in the scenario of Section II can be detected and adapted with the tools of our adaptation framework. A **sink behavior** is observed when a node persists in accepting tasks from other nodes however prefers to work on tasks of certain neighbors, or under-performs in task processing. This behavior is recognizable by a dense delegation of tasks to the sink possibly requiring different capabilities and a low task completion notification in the observed time span. In the notion of Groovy scripts introduced in Section IV, Listing 3 shows the procedure used to detect and adapt nodes with sink behavior in the testbed framework.

```
// in the monitoring loop
def sinkNode = env.triggerSink(4) // sink trigger with threshold 4 tasks
if (sinkNode) { // sink suspected
  if (env.analyzeTaskQueueBehavior(sinkNode)) { // analyze task history
    def simNodes = sim.getSimilar(sinkNode) // call similarity service
    altNodes = []
    simNodes.each { s ->
      if (env.loadTolerable(s))
      altNodes += s // find nodes with tolerable load
    }
    def neighborNodes = env.getNeighbors(sinkNode) // affected neighbors
    neighborNodes.each { n ->
      n.dStrat = { refs -> // overwrite dStrat from Listing 1.
        refs += altNodes // add alternatives channels
        refs -= sinkNode // remove channel to sink
        ... // selection strategy
      }
    }
  }
}
```

Listing 3. Code example for sink adaptation.

The script extract defines the task queue trigger's `triggerSink` threshold first. If the limit of four tasks is violated by a node analysis `analyzeTaskQueueBehavior` scans the affiliated task history and compares the latest delegation and task status reporting patterns of the node. If a sink is detected, the *Similarity Service* `sim` is called and returns a set `simNodes` of possible candidates for replacement. In the next loop the candidates' current task queue size is examined (`loadTolerable`). Only those with few tasks are added to the final alternative nodes `altNode` list. In the last step the delegation strategies of the neighbors of the sink node are updated. The alternatives are added to the possible delegation candidates and the `sinkNode` is avoided.

A moderate use of queue capacity in contrast to high and

exceeding delegation rates despite available alternatives causes overload at single nodes. This identifies the **factory behavior**. Again interaction data uncovers the misbehavior expressed by a high fluctuation of tasks from the factory and a low task completion rate in the monitored interval. The Groovy script in Listing 4 presents our factory adaptation algorithm for the testbed framework.

```
1  // in the monitoring loop
2  def factoryNode = env.triggerFactory(2) // factory trigger with threshold 2 tasks
3  if (factoryNode) { //factory suspected
4    if (env.analyzeDelegationBehavior(factoryNode)) {// analyze task history
5      def simNodes = sim.getSimilar(factoryNode) // call similarity service
6      altNodes = []
7      simNodes.each { s ->
8        if (env.loadTolerable(s))
9          altNodes += s // find nodes with tolerable load
10     }
11     def neighborNodes = env.getDelegator(factoryNode) // affected delegators
12     neighborNodes.each { n ->
13       n.dStrat = { refs -> // overwrite dStrat from Listing 1.
14         refs += altNodes //add alternatives channels
15         refs -= factoryNode // remove channel to factory
16         ... // selection strategy
17       }
18     }
19     factoryNode.dStrat={} // no delegations allowed
20   }
21 }
```

Listing 4. Code example for factory adaptation.

The factory trigger's threshold `triggerFactory` fires diagnosis on task queue sizes below two tasks. If `analyzeDelegationBehavior` confirms a pattern with high delegation frequency a factory node is detected. The same as with a sink, a selection of alternative nodes for a factory node replacement is collected. From this list only those with minor load are further considered. Then the affected neighbors who are delegating nodes (`getDelegators`) are freed from the factory and provided with the alternative nodes. Finally, the delegation strategy of the delegating neighbors is adapted. In contrast to the sink in the last step all factory's delegation channels are closed temporarily.

VI. Experiments

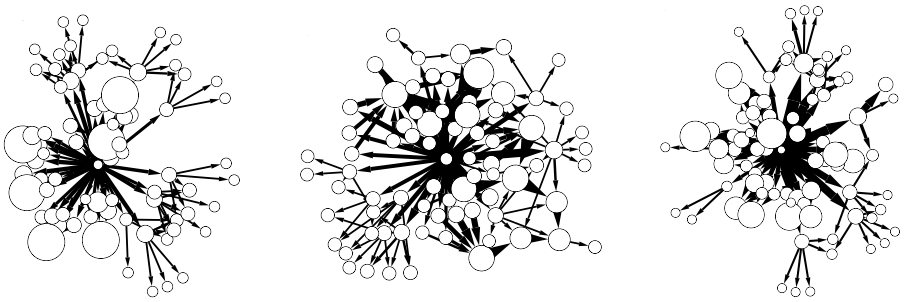
In our experiments we evaluate the efficiency of similarity based adaptation in a virtual team of a crowd of task-based services. This team comprises a few hundreds of collaborators. The assumption is that some of the HPSs expose a misbehavior with the progress of time. Misbehavior is caused by team members that for various reasons including, e.g., task assignment overload, change of interest, or preference for particular tasks, start to process assigned tasks irregularly. Our strategy is to detect misbehavior by analyzing the task processing performance of the team. A degrading task processing rate indicates misbehavior. The main idea is to detect these degradations, identify the misbehaving team members with a task history analysis, and, in time, provide a fitting replacement for the misbehaving member. This member match is provided by our *Similarity Service* that mines the capabilities and noted changes at the members The main information source of our misbehavior analysis and detections is the data contained in the delegated tasks.

A. Scenario Overview

Following the concept of crowdsourcing we modeled a scenario showcasing the interaction dynamics of a specific sector comprised by a bunch of teams. Interested parties wish to outsource multiple tasks to a crowd. In order to get their tasks completed they refer to an entry point service that forwards tasks to multiple teams of the crowd. A team comprises two types of members. The first, the delegators, receive new tasks directly from the entry point. Instead of working on the tasks their concern is to redistribute the tasks to their neighbors. These neighbors are also called workers. A delegator picks its most capable and trusted workers that can process the assigned task. Each team is specialized on a particular type of task. Tasks carry keyword information in order to distinguish which team receives a particular task.

A task's life-cycle starts at the entry point that provides the team constantly with new tasks. It acts as a proxy between team and actual task owner and its main assignment is to decide which of the team members is suitable for processing. The question is how to find the appropriate worker for a task. All services are registered at startup by the registry including their capabilities. Though, the information of the registry remains static and becomes outdated over the course of time. Members' processing behaviors can change over time when tasks start to be delegated and processing loads vary. Thus, the entry point can refer to the environment's registry for candidates at the beginning and shortly after bootstrapping but once profiles start to change the lookup information becomes inaccurate. The solution is the *Similarity Service* which is aware of these changes. It tracks the interest shift by monitoring the delegation behavior between interacting neighbors. Therefore, the service provides the most accurate candidates for a delegation during runtime. However, at the contrary the *Similarity Service* cannot provide satisfying results from the beginning because of the lack of interaction data.

Once the appropriate candidate is selected by the entry point it delegates the task. Teams, as in our scenario are composed of a sub-community of HPSs that know and trust each other and, hence, keep references to each other in a neighbor-list. Delegations in the team are only issued between these trusted neighbors. Tasks are associated with a deadline to define the task's latest demanded completion time and a processing effort. Each worker has its individual task processing speed depending on the knowledge compared to the tasks requirements and the current work load. At the end of a task's life-cycle, a worker reports the task as complete, or if the deadline is missed, expired. The main focus of the misbehavior regulation is to avoid tasks to expire. Our algorithm identifies failing services by observing the task throughput. It filters tasks that missed their deadline in a certain periode. Such a misbehavior is then adapted with the help of the knowledge of the *Similarity Service* and the task history. First the most similar members to the misbehaving are selected and than with a task queue size



(a) No adaptation applied. (b) Adaptation through mirroring. (c) Adaptation through teleportation.

Fig. 4. Evolving interaction networks based on adaptation actions.

analysis the least loaded chosen for an adaptation. Depending on the current trust-based adaptation strategy channels between working nodes are added or delegations shifted to competent but less busy workers.

B. Experiment Setup

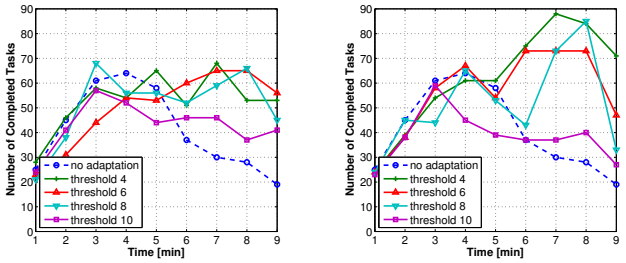
In order to simulate described medium size teams of the aforementioned crowdsourcing model, we set up following environment. The teams comprise a total of 200 collaborators represented by Web services created by G2 scripts deployed to one backend instance. 20% of these members expose a delegation behavior the rest works on assigned tasks. All services are equipped with a task queue. As in the real world the services are not synchronized and have their individual working slots. Usually a worker processes one entire task per slot. A worker starts to misbehave once its task queue is filled past the threshold of 6 tasks. It then reduces its working speed to one third. A total of 600 task are assigned to the environment. We do not adapt from start. At start there is a period of 200 task with no adaptation. Then in an adaptation cycle the workers task queue size is monitored by tracing the delegation flow among the nodes. The difference between acknowledged assignments and complete or expired reported tasks results in the current task queue size at a particular worker. Once this number exceeds the preset task queue threshold which we vary for the different results of our experiments, the similarity service is invoked for a list of workers with similar capabilities. In a loop over this list sorted by best match the candidate is picked with the currently smallest task queue size. The applied adaptation action depends on the experiment's current adaptation strategy. In trust mirroring a channel between two similar workers is opened which allows the overloaded node additionally to delegate one task per slot over the new channel. In trust teleportation the overloaded worker is relieved from the most delegating neighbor and a new channel is opened from the delegator to a substitute worker.

Figure 4 shows the temporal evolution of dynamic interactions under different adaptation actions. It demonstrates the changes in interactions for a threshold of 6 tasks in the three sub-figures. A node's size represents the total number of incoming delegations. Larger edges indicate a high number of delegations across this channel with the arrow pointing in the delegation direction. Therefore, the node in the middle is easily identified as the entry point. It sheer provides tasks to all the connected delegators. Figure 4(a) shows that these delegators prefer selected workers to complete their tasks. In this figure six extremely overloaded workers are present after the first 200 tasks have left the entry point. Only a few others are sporadically called. Figure 4(b) represents the effects at the end of the experiment for the mirroring strategy. The effects of this strategy are clearly visible. The load between the workers is better distributed. A few, however more equilibrate worker nodes remain compared to no action because the delegators still prefer to assign tasks to their most trusted workers. However, a lager number of new workers is added at the outer leaves of the tree which release these nodes from their task load. Figure 4(c) highlights the situation with the trust teleportation strategy. The side-effects here show that the number of loaded nodes remains almost the same. However, the load peek at the preferred workers is kept below the predefined threshold. Once exceeded the worker is relieved from its delegator and a replacement found. With this strategy workers get loaded to their boundary and are then replaced with new workers.

In our experiments we tested the effectiveness of adaptations with different task queue threshold triggers. The effectiveness is measured by the total task processing performance at the end of the experiment. Only completely processed and reported tasks went into the final result.

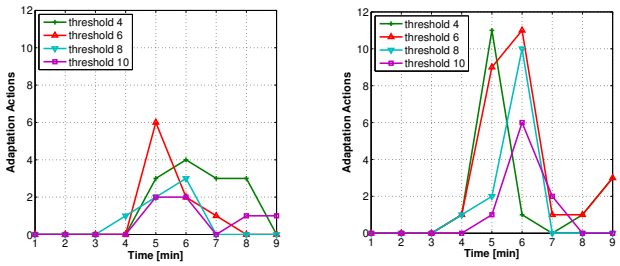
C. Result Description

Figure 5 presents the results of our simulation evaluations. Both diagrams provide the time-line in minutes on the x-axis and the number of completed tasks at the end of this period on the y-axis. In both cases there is a well noticeable incrementation of completed tasks until minute 4. This is when the first 200 tasks have been distributed to the workers. The task distribution is not linear over the measured period. This is due to the fact that at the beginning not so many tasks can be distributed because of bootstrapping delays in the G2 backend. This is also when the first adaptations are deployed. Whilst the task completion ratio decreases rapidly at this point if no adaptation actions are taken (demonstrated by the dashed line) the other lines represent the progress of the task completion when different thresholds triggers together with reconfigurations are applied. The diagrams in Figure 6 show again the time-line on the x-axis and the number of applied actions at the end of the period on the y-axis.



(a) Mirroring. (b) Teleportation.

Fig. 5. Adaptations using different thresholds for mirroring and teleportation.



(a) Actions applied in mirroring. (b) Actions applied in teleportation.

Fig. 6. Number of adaptation actions applied using different strategies.

Figure 5(a) details the results of an adaptation strategy using trust mirroring. Generally all strategies perform better than when no action is taken. With a trigger threshold of 4 tasks and approximately 3 actions every minute the curve exposes an increment followed by a decrement between 70 and 50 completed task every minute. The pattern is similar to the curve representing a threshold of 8. Figure 6(a) shows that the adaptations are less and the altering of direction in Figure 5(a) is slower. The smoothest adaptations result from a trigger matching the real worker's threshold of 6 tasks. Comparing the figures, a smaller growth of success in task completion is noticed after the deployment of the 3 followed by 4 adaptations between minute 4 and 6. A threshold of 10 tasks decreases slower than an adaptation free environment but with only about 20 more successfully processed tasks. With the same adaptation effort as at threshold 8 this strategy exposes an overall inconvenient timing of the adaptations and can be considered impractical. The situation is different in Figure 5(b). As Figure 6(b) shows, there are more adaptations deployed with this strategy. But not without leaving following side-effects. The curve of adaptations triggered at threshold 4 increases rapidly after minute 5 when a total of 11 new channels are provided to new workers in a time slot of 1 minute. Even if again with the smoothest progress among the successful strategies the curve representing actions at threshold 6 cannot reach the top performances of their neighbors (threshold 4 and 8). Instead the 20 new channels set between minute 4 and 6 let the system performance progress even. Finally the curve of threshold 10 has a noticeable regress between minute 3 and 4 caused by the dynamics of the system. In the following this type of strategy with only 9 adaptations in total is not able to recover and is even outperformed by the no adaptation run. The final results show that the precise timing of multiple adaptations in a short term is most convenient for environment adaptation actions. However this has a trend to highly altering task processing results (e.g., approximately 40 task for a threshold 8 in Figure 5(b)). Comparing both, a strategy where the trigger matches the environments actor's threshold of 6 is most practical in a balanced environment. Strategies with a threshold above 8 are infeasible for this setup. Generally the teleportation strategy performs better than mirroring, however requires the double and more adaptation actions.

VII. Related Work

Two main research directions on **self-adaptive properties** emerged in the past years. One initiated by IBM and presented by the research of autonomic computing [19], [20] and the other manifested by the research on self-adaptive systems [7]. Whilst autonomic computing includes research on all possible system layers and an alignment of self-* properties to all available system parts, self-adaptive system research pursuits a more global and general approach. The efforts in this area focus primarily on research above the middleware layer and consider self-* methodologies that adapt the system as a whole. These include higher layers such as models and systems' architecture [21], application layer, and in particular interesting for our research are large-scale agent-based systems [22] , Web services , and their orchestration [23]. Self-adaptive ideas can be found for middleware [24] and also at a lower layer include, e.g., operating systems [25]

With current systems growing in size and ever changing requirements plenty of challenges remain to be faced such as autonomic adaptations [26] and service behavior modeling [27]. The self-adaptive research demonstrated in this paper strongly relates to the challenges in Web services and workflow systems. Apart from the cited, substantial research on self-adaptive techniques in Web Service environments has been conducted in the course of the European Web service technology research project WS-Diamond (Web-Service DIAgnosinbility, MONitoring and Diagnosis). The recent contributions focus in particular on QoS related self-adaptive strategies and adaptation of BPEL processes [28], [29]. Others are theoretical discussions on self-adaptive methodologies [30].

Regarding **runtime evaluation**, several approaches have been developed which could be applied for testing adaptation mechanisms. SOABench [31] and PUPPET [32], for instance, support the creation of mock-up services in order to test workflows. However, these prototypes are restricted to emulating non-functional properties (QoS) and cannot be enhanced with programmable behavior. By using Genesis2 [4] which allows to extend testbeds with plugins we were able to implement a testbed which was flexible enough to test diverse adaptation mechanisms.

Human-Provided Services [3] close the gap between Software-Based Services and humans desiring to provide their skills and expertise as a service in a collaborative process. Instead of a strict predefined process flow [33], these systems

are denoted by ad-hoc contribution request and loosely structured processes collaborations. The required flexibility induces even more unpredictable a system property responsible for various faults. In our approach we monitor failures caused by misbehavior of service nodes. The contributed self-adaptive method recovers by soundly restricting delegation paths or establishing new connections between the nodes.

Over the last years, **trust** has been defined from several points of views [13], however, until now, no agreed definition exists. Unlike the area of network and computer security we focus on the notion of dynamic trust from a social perspective [12]. Our notion of trust [10] is based on the interpretation of collaboration behavior [10], [13] and dynamically adapting skills and interest similarities [14], [15]. In the introduced environment we make explicit use of the latter one.

VIII. Conclusion and Outlook

The main objective of this work was to demonstrate the successful integration of two frameworks. On one side the G2 [4] SOA testbed and on the other the extensible VieCure [2] adaptation framework. The two remain separate and independent frameworks and are only loosely coupled. As a first extension in this paper we added to the adaptation loop a module providing similarity ratings for the testbed services. The results of our evaluation confirm that the deployed task processing team scenario and the two adaptation strategies trust mirroring and teleportation interplay satisfactorily. A precise timing and a careful aligned threshold for the actions is essential to reach high amounts of task completion rates. This observation emphasizes our attempt in implementing non-intrusive self-healing recovery strategies that can not always relate on accurate status information for a decision.

In our future work we plan to deploy a whole crowdsourcing environment with miscellaneous teams to a distributed testbed. It will then also become essential to distribute and duplicate some of the components of the adaptation framework, e.g., logging, diagnosis and analysis modules. We plan a layered adaptation strategy that provides an interface to deploy local adaptations and allows global adaptations on a higher layer involving utility based changes for the whole crowd. New models of Mixed System's misbehavior and extended rules for detection and diagnosis of behavior will become necessary.

Acknowledgment

This work received funding from the EU FP7 programme under the agreements 215483 (SCube) and 216256 (COIN).

References

[1] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Trans. on Softw. Eng.*, vol. 30, pp. 311–327, 2004.

[2] H. Psaier, F. Skopik, D. Schall, and S. Dustdar, "Behavior Monitoring in Self-healing Service-oriented Systems," in *COMPSAC*. IEEE, 2010.

[3] D. Schall, H.-L. Truong, and S. Dustdar, "Unifying human and software services in web-scale collaborations," *Internet Computing*, vol. 12, no. 3, pp. 62–68, May-June 2008.

[4] L. Juszczyk and S. Dustdar, "Script-based generation of dynamic testbeds for soa," in *ICWS*. IEEE Computer Society, 2010.

[5] R. Sterritt, "Autonomic computing," *Innovations in Systems and Software Engineering*, vol. 1, no. 1, pp. 79–88, 2005.

[6] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, no. 1, pp. 5–18, 2003.

[7] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[8] M. Vukovic, "Crowdsourcing for Enterprises," in *Proceedings of the 2009 Congress on Services*. IEEE Computer Society, 2009, pp. 686–692.

[9] D. Brabham, "Crowdsourcing as a model for problem solving: An introduction and cases," *Convergence*, vol. 14, no. 1, p. 75, 2008.

[10] F. Skopik, D. Schall, and S. Dustdar, "Modeling and mining of dynamic trust in complex service-oriented systems," *Information Systems*, vol. 35, no. 7, pp. 735–757, 11 2010.

[11] S. Dustdar, "Caramba—a process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams," *Distrib. Parallel Databases*, vol. 15, no. 1, pp. 45–66, 2004.

[12] C.-N. Ziegler and J. Golbeck, "Investigating interactions of trust and interest similarity," *Dec. Sup. Syst.*, vol. 43, no. 2, pp. 460–475, 2007.

[13] T. Grandison and M. Sloman, "A survey of trust in internet applications," *IEEE Communications Surveys and Tutorials, 2000,*, vol. 3, no. 4, 2000.

[14] J. Golbeck, "Trust and nuanced profile similarity in online social networks," *ACM Trans. on the Web*, vol. 3, no. 4, pp. 1–33, 2009.

[15] Y. Matsuo and H. Yamamoto, "Community gravity: Measuring bidirectional effects by trust and rating on online social networks," in *WWW*, 2009, pp. 751–760.

[16] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Proc. and Mgmt.*, vol. 24, no. 5, pp. 513–523, 1988.

[17] Groovy Programming Language, http://groovy.codehaus.org/.

[18] WS-Addressing, http://www.w3.org/Submission/ws-addressing/.

[19] IBM, *An architectural blueprint for autonomic computing*. IBM White Paper, 2005.

[20] R. Sterritt, "Autonomic computing," *ISSE*, vol. 1, no. 1, pp. 79–88, 2005.

[21] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *SEAMS*, 2006, pp. 2–8.

[22] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, I. W. N. Mills, and Y. Diao, "Able: A toolkit for building multiagent autonomic systems," *IBM Syst. J.*, vol. 41, no. 3, pp. 350–371, 2002.

[23] L. Baresi and S. Guinea, "Dynamo and self-healing bpel compositions," in *ICSE*, 2007, pp. 69–70.

[24] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas, "Reflection, self-awareness and self-healing in openorb," in *WOSS*, 2002, pp. 9–14.

[25] M. W. Shapiro, "Self-healing in modern operating systems," *ACM Queue*, vol. 2, no. 9, pp. 66–75, 2005.

[26] J. O. Kephart, "Research challenges of autonomic computing," in *ICSE*, 2005, pp. 15–22.

[27] K. Kaschner and K. Wolf, "Set algebra for service behavior: Applications and constructions," in *BPM*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 193–210.

[28] R. Halima, K. Guennoun, K. Drira, and M. Jmaiel, "Non-intrusive QoS Monitoring and Analysis for Self-Healing Web Services," in *ICADIWT*, 2008, pp. 549–554.

[29] R. Halima, K. Drira, and M. Jmaiel, "A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services," in *ICWS*, 2008, pp. 104–111.

[30] M. Cordier, Y. Pencolé, L. Travé-Massuyès, and T. Vidal, "Characterizing and checking self-healability," in *ECAI*, 2008, pp. 789–790.

[31] D. Bianculli, W. Binder, and M. L. Drago, "Automated performance assessment for service-oriented middleware," Faculty of Informatics - University of Lugano, Tech. Rep. 2009/07, November 2009. [Online]. Available: http://www.inf.usi.ch/research_publication.htm?id=55

[32] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," in *TestCom/FATES*, ser. Lecture Notes in Computer Science, vol. 5047. Springer, 2008, pp. 266–282.

[33] F. Leymann, "Workflow-based coordination and cooperation in a service world," in *CoopIS, DOA, GADA, and ODBASE*, 2006, pp. 2–16.