2010 34th Annual IEEE Computer Software and Applications Conference

# Behavior Monitoring in Self-healing Service-oriented Systems

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

***Abstract*—Web services and service-oriented architecture (SOA) have become the de facto standard for designing distributed and loosely coupled applications. Many service-based applications demand for a mix of interactions between humans and Software-Based Services (SBS). An example is a process model comprising SBS and services provided by human actors. Such applications are difficult to manage due to changing interaction patterns, behavior, and faults resulting from varying conditions in the environment. To address these complexities, we introduce a self-healing approach enabling recovery mechanisms to avoid degraded or stalled systems. The presented work extends the notion of self-healing by considering a mixture of human and service interactions observing their behavior patterns. We present the design and architecture of the VieCure framework supporting fundamental principles for autonomic self-healing strategies. We validate our self-healing approach through simulations.**

***Keywords*-Self-healing model, monitoring, recovery, mixed service-oriented system, delegation behavior**

## I. Introduction

Large-scale distributed applications become increasingly dynamic and complex. Adaptations are necessary to keep the system fit and running. New requirements and flexible component utilization call for updates and extensions. Thus, a challenge is the sound integration of new and/or redesign of established components. Integration must also consider changing dependencies. Unfortunately, to cope with all efforts, including deployment, integration, configuration, and fine tuning, monitoring and control of the system has proven sheer impossible by humans alone [1]. Today's SOAs are composed of loosely coupled services orchestrated to collaborate on various kinds of tasks. However, their benefit, modularity and an almost infinite number of combinations, fosters unpredictable behavior and as a consequence results in poor manageability. *Mixed Systems* extend the solely software implemented capabilities of traditional Service-oriented Systems with human provided services. The integration of humans and software-based services is motivated by the difficulties to adopt human expertise into software implementations. Rather than dispense with the expertise, in *Human-Provided Services (HPSs)* a human handles tasks [2] behind a traditional service interface. The mix of common services based purely on software denoted as *Software-Based Service (SBS)* and HPS forms a Mixed System.

Systems with self-healing properties are part of the Autonomic Computing [1] and Self-adaptive Systems [3] research. The self-healing properties of a system enhance new or existing unpredictably, unsatisfactorily manageable environments with self-aware recovery strategies. Hence, self-healing is considered a property of a system that comprises fault-tolerant, self-stabilizing, and survivability capabilities, and on exceptions, relies also on human intervention [4], [5]. A certain self-awareness is guaranteed by a continuous flow of status information between self-healing enhancement and environment. Inherited from fault-tolerant systems, the success of self-healing strategies depends on the recognition of the system's current state.

### A. Self-healing principles

Mixed Systems are designed and built for long term use. Once available they are expected to remain accessible and tend to grow in size. To keep the system prevalent new services are integrated and legacy ones are updated. New requirements, advances in and novel technologies involve necessary changes. Therefore, a certain adaptability is required and expected from the system. However, the required flexibility increases the complexity of the system, and adaptations possibly cause unexpected behavior. The main goal of a self-healing approach is to avoid unpredictable behavior leading to faults. Filtered events are correlated to analyze the health of the system. The problem is identified and appropriate recovery actions are deployed [6]. The current health is usually mapped to recognizable system states as provided by the generic three state model for self-healing as for example discussed in [4].

According to their classification a system is considered in $healthy\ state$ when not compromised by any faults. Once a degradation of system performance caused by faults is detected, the system moves to a $degraded\ state$ but still functions. The situation is in particular observed in large-scale systems. This provides self-healing extensions with time for carefully planned recovery strategies that do not only include fault recovery by repair actions, but also sound deployment and compensation of side-effects. Finally, if the faults affect essential parts or a majority of the nodes the system's behavior becomes unpredictable and ultimately stalls. The system is considered in $unhealthy\ state$.

0730-3157/10 $26.00 © 2010 IEEE
DOI 10.1109/COMPSAC.2010.43

Self-healing tries to avoid a stalled system. The state is prevented by a combination of self-diagnosing and self-repairing capabilities [3]. A compelling precondition for any self-healing enhancement is a continuous data-flow between those and the guarded system. According to [1] a control loop is the essence of automation in a system. In detail [7] presents the autonomic manager as a generic layout for any self-management property, including self-healing. The manager relies on a control loop and includes monitor, analyze, plan, and execute modules.

### B. Contributions

Possible fault sources in Mixed Systems are manifold. Failures occur on all layers including the infrastructure layer, e.g., hardware and communication channels, implementation, such as mistakes and errors in application software, and application layer, due to errors in utilization and incomprehensible administration. In this work we focus on a novel kind of fault source: unpredictable and faulty behavior of services in a Mixed System. For that purpose, we observe the behavior of the heterogeneous services and their interactions. In particular, we focus on task delegation behavior in a collaborative scenario. Services have a limited buffer for tasks and excessive delegations to single nodes in the network can cause buffer overloads, and furthermore, may lead to service degradation or ultimately to failure. It is thus essential that we identify misbehavior, analyze the cause, and heal the affected services. Moreover, we use a non-intrusive healing approach which punishes misbehavior by protecting affected nodes from load and restricting the delegation options of misbehaving nodes.

In this paper we present the following contributions:

- *Delegation Behavior Models.* We identify the fundamental delegation behavior models and their effects on the health state of the network.
- *Failure Models.* We outline failure models in the system caused by misbehavior and analyze their root cause.
- *VieCure Architecture.* We present our self-healing framework using state of the art Web services technologies.
- *Recovery Strategies.* We formulate algorithms to compensate the effects of misbehavior and facilitate fast system recovery.
- *Evaluation.* We simulate discussed recovery strategies to enable sophisticated self-healing in mixed service-oriented networks.

The rest of the paper is structured as follows. In Section II we outline our motivation for the chosen approach, give a guiding example scenario, and identify two types of misbehavior. Sections III and IV describe the components and architecture and detail our self-healing framework. The algorithm presented in Section V represents our misbehavior healing approach. An evaluation with experiments follows in Section VI. Related work is discussed in Section VII, and the paper is concluded in Section VIII.

## II. Flexible Interactions and Compositions

In this section we introduce a cooperative system environment, explain the motivation for our work, and deal with the major challenges of self-healing in mixed SOA.

### A. Scenario

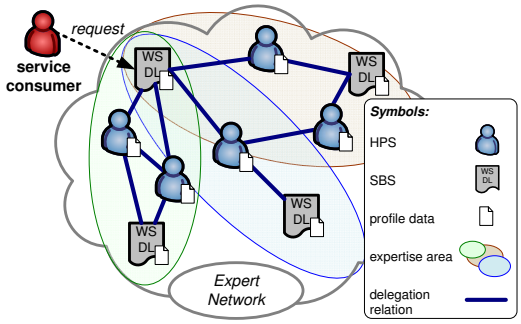
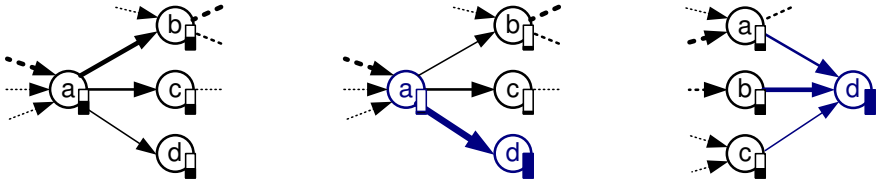



Figure 1. Flexible cooperation of actors in an expert network.

Today, 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. For a single service consumer this scenario is shown in Figure 1.

We model a mixed expert network consisting of HPSs [2] and SBSs that belong to different communities. The members of these communities are discovered based on their main expertise areas (depicted as shaded areas), and are connected through certain relations (see later for details). Community members receive requests from external service consumers, process them and respond with appropriate answers. 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 *Expert Network*). While the results of certain simple but often repeated experiments can be efficiently processed by SBSs, analyzing more complex data usually needs human assistance. For that purpose, HPS offers the advantage of loosely coupling and flexible involvements of human experts in a service-oriented manner. Therefore, our environment uses standardized SOA infrastructures, relying on widely adopted standards, such as SOAP and the Web Service Description Language (WSDL), to unify humans and software services in one harmonized environment.

Various circumstances may be the cause for 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




(a) Healthy behavior. (b) Delegation factory. (c) Delegation sink.

Figure 2. Delegation behavior models.

idle. Load distribution problems can be compensated with the means of *delegations* [8]. Each expert in a community knows (i.e., realized as 'knows' relation in FOAF profiles[1]) some 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 tasks can a community member accept without neglecting other work? However, if misbehavior cannot be avoided in the network, its effects need to be compensated.

Consider the following scenario: Someone is invited to join a community, e.g., computer scientists, in the expert network. Since she/he is new and does not know many other members, she/he is not well connected in the Web. In the following, she/he will receive tasks that match her/his expertise profile, but is not able to delegate to other members. Hence, she/he may get overloaded if several tasks arrive in short time spans. A straightforward solution is to find another member with similar capabilities that has free capacities. A central question in this work is how to support this process in an effective manner considering global network properties. In this paper we focus on failures in the ad-hoc expert network. Such failures impact the network in a harmful manner by causing degradations. In particular, we deal with misbehavior of community members and highlight concepts for *self-healing* to recover from degraded states in SOA-based environments comprising human and software services.

### B. Delegation Behavior

Each node, i.e., community member, has a pool of open tasks. Therefore, the load of each node varies with the amount of assigned tasks. In Figure 2 the load of nodes is depicted by vertical bars. If a single node cannot process assigned tasks or is temporarily overloaded, it may delegate work to neighbor nodes. The usual delegation scenario is shown in Figure 2(a). In that case, node $a$ delegates work to its partner nodes $b$, $c$, and $d$, which are connected by channels. A channel is an abstract description of any kind of link that can transport various information of communication, coordination and collaboration. In particular, a *delegation channel* has a certain *capacity* that determines the amount of tasks that may be delegated from a node $a$ to a node $b$ in a fixed time frame. None of the nodes is overloaded with work in the healthy state.

**Delegation Factory.** As depicted in Figure 2(b) a delegation factory produces unusual amounts (i.e., unhealthy) of task delegations, leading to a performance degradation of the entire network. In the example, node $a$ accepts large amounts of tasks without actually performing them, but simply delegates to its neighbor node $d$. Hence, $a$'s misbehavior produces high load at this node. 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.

**Delegation Sink.** A delegation sink behaves as shown in Figure 2(c). Node $d$ accepts more task delegations from $a$, $b$, and $c$ as it is actually able to handle. In our collaborative network, this may happen due to the fact that $d$ either underestimates the workload or wants to increase its reputation as a valuable collaboration partner in a doubtful manner. Since $d$ is actually neither able to perform all tasks nor to delegate to colleagues (because of missing outgoing delegation channels), 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.

Healing refers to compensating the effects of delegation misbehavior by adapting structures in the delegation network. This includes modifying the capacity of delegation channels, as well as adding new channels and removing existing ones.

## III. Architecture Overview

One of the biggest challenges in Mixed Systems is to support flexible interactions while keeping the system within boundaries to avoid degraded or stalled system states. Thus, adaptation mechanisms are needed to guide and control interactions. In this section we introduce the *VieCure* framework to support self-healing principles in mixed service-oriented systems. Such environments demand for additional

[1]FOAF: http://xmlns.com/foaf/spec/

tools and services to account for human behavior models and complex interactions. In the following, we present the overall architecture, inspired by existing architectural models in the self-healing and autonomic computing domain, and introduce novel components such as a *behavior registry* holding information regarding HPS delegation behavior.

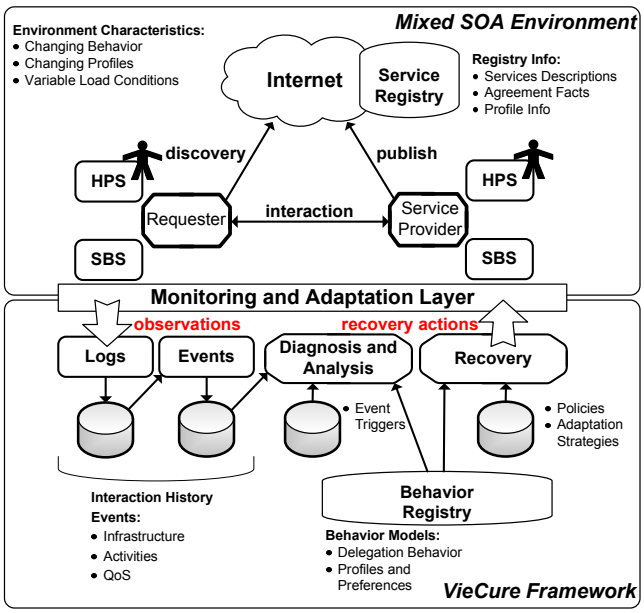



Figure 3. Environment overview and the *VieCure* framework.

Figure 3 shows the overall framework model comprising three main building blocks: *SOA Environment* consisting of human and software services, *Monitoring and Adaptation Layer* to observe and control the actual environment and the *VieCure* framework providing the main features to support self-healing actions.

### A. Mixed SOA Environment

Many collaboration and composition scenarios involve interactions spanning human actors as well as software services. Traditional SOA architectures were designed to host SBSs without considering Human-Provided Services.

We extend the architectural model by introducing:

- A service registry maintaining information related to human and software services.
- Definition of interaction patterns and interaction constraints using Web service technology.
- Enhanced service-related information by describing human characteristics and capabilities.

The resulting *environment characteristics* are dynamic, because of changing behavior and profiles, and the need for adaptation mechanism due to variable load conditions (e.g., changing availability of human actors and changing amount of task that need to be processed).

### B. Monitoring and Adaptation Layer

The main building block of an environment enhanced with self-* capabilities is a *feedback loop* enabling adaptation of complex systems. The functions of a feedback loop can be realized as a *MAPE-K* cycle (Monitor, Analyze, Plan, Execute, and K denoting the Knowledge) [7]. Therefore our architecture needs to integrate the functions of this loop by performing two essential steps:

**Observations.** Part of the knowledge base is provided by observations. Observations constitute most of the current knowledge of the system. Interaction data is gathered from the mixed system environment and stored in the logging database (denoted as *Logs*). *Events* are registered and captured in the environment, stored in historical logs, and serve as input for triggers and the diagnosis.

**Recovery Actions.** By filtering, analyzing, and diagnosing events, an adaptation may need to be performed. Recovery actions are parts of a whole adaptation plan determined by diagnosis. Single recovery actions are deployed in correct order and applied to the environment by *Recovery* module.

## IV. VieCure Framework

The building blocks of the *VieCure* framework are be detailed in this section. Figure 4 shows the fundamental interplay of VieCure's components. The *Monitoring and Adaptation Layer* is the interface to the controlled environment that is observed by the framework and influenced afterward through corrective actions. All monitored interactions, such as SOAP-based task delegations (see Listing 1), are stored for later analysis by *Interaction Logging Facilities*. Environment events, including adding/removing services or state changes of nodes, are stored by similar *Event Logging Facilities*. Logs, events, and initial environment information represent the aggregated knowledge used by the *VieCure* framework to apply self-healing mechanisms. The effectiveness and accuracy of the healing techniques strongly depend on data accuracy.

The *Event Monitor* is periodically scheduled to collect recent interactions and events from the logging facilities. Upon this data, the monitor infers higher level composite events ($c - event$). Pre-configured triggers for such events, e.g. events reporting agreement violations, inform the *Diagnosis Module* about deviations from desired behavior. Furthermore, the actual interaction behavior of nodes is periodically updated and stored in the *Behavior Registry*. This mechanism assists the following diagnosis to correlate behavior changes and environment events. Furthermore, profiles in conjunction with the concept of HPSs allow to categorize these services and determine root causes.

Once a deviation indicating composite event triggered the *Diagnosis Module*, a root cause analysis is initiated. Previously captured and filtered interaction logs as well as actual node behaviors, assist a sophisticated diagnosis and to recognize the mixed system's health state. On failures a set

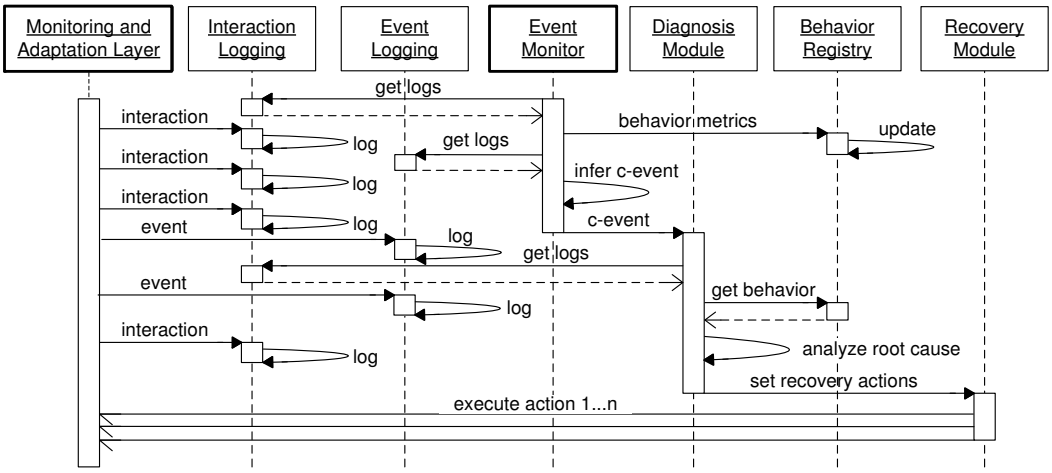



Figure 4. VieCure's fundamental mode of operation.

of corrective recovery actions is submitted to the *Recovery* module.

A substantial part of recovery is the self-healing policy registry (underneath the *Recovery* block in Figure 3). It manages available adaptation methods. As mentioned before, adaptations and constraints applied by self-healing policies include, for example, boundaries and agreements imposed on the services defining the interaction paths and limiting recovery strategies. The recovery module executes the recovery actions and influences the mixed system environment through the *Monitoring and Adaptation Layer*.

### A. Interaction Monitoring

Interactions between community members of the expert network are modeled as standardized SOAP messages with header extensions (see also [8]), as shown in Listing 1.

```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
 xmlns:vietypes="http://www.infosys.tuwien.ac.at/Type"
 xmlns:hps="http://myhps.org/Type"
 xmlns:hpsht="http://myhps.org/HumanTask"
 <soap:Header>
  <wsa:MessageID>uuid:722B1240−...</wsa:MessageID>
  <wsa:ReplyTo>http://www.expertweb.org/Actor#Harald</wsa:ReplyTo>
  <wsa:From>http://www.expertweb.org/Actor#Harald</wsa:From>
  <wsa:To>http://www.expertweb.org/Actor#Florian</wsa:To>
  <wsa:Action>http://myhps.org/Action/Delegation</wsa:Action>
  <vietypes:activity url="http://www.expertweb.org/Activity#42"/>
  <vietypes:delegation hops="3"/>
  <vietypes:timestamp value="2010-01-29T15:13:21"/>
  <hpsht:taskContext>
    <hpsht:deadline="2010-01-30T12:00:00"/>
    <hpsht:priority>...</hpsht:priority>
  </hpsht:taskContext>
 </soap:Header>
 <soap:Body>
   <hps:prepReport>
     <prepReport:requ>Please create a report for experiment X</prepReport:requ>
     <prepReport:generalterms>algorithm</prepReport:generalterms>
     <prepReport:keywords>ranking, interactions, graph</prepReport:keywords>
     <prepReport:resource url="http://.../experimentX"/>
   </hps:prepReport>
 </soap:Body>
</soap:Envelope>
```

Listing 1. Simplified interaction example.

A logging service is part of the monitoring layer to capture all interactions performed in the network. Header extensions include the context of interactions (i.e., the activity that is performed), delegation restrictions (e.g., the number of hops), identify the sender and receivers with WS-Addressing[2], and hold some meta-information about the activity type itself. For HPS, SOAP messages are mapped to user interfaces by the HPS framework [2]. *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.

### B. Event Trigger, Diagnosis and Recovery Actions

The event monitor is an integral part of the monitoring layer. As previously described it constantly logs arriving events from the environment and composes log and event history to higher level events. Events from the environment are delivered by a reliable and asynchronous event bus provided by the Java Message Service (JMS)[3].

```
<complexType name="Event"> <sequence>
 <element name="logSeqNumber" type="int"></element>
 <element name="logTime" type="time"></element>
 <element name="eventSeqNumber" type="int"></element>
 <element name="eventTime" type="time"></element>
 <element name="eventOrigin" type="string"></element>
 <element name="eventType" type="tns:EventType"></element>
 <element name="eventExtendedType" type="tns:ExtEventType">
 </element>
 <element name="eventDescription" type="string"><element>
 <element name="eventSeverity" type="tns:Severity"></element>
  ...
 </sequence>
</complexType>
```

Listing 2. Extract of event specification.

[2]http://www.w3.org/Submission/ws-addressing/
[3]http://java.sun.com/products/jms/

The structure of an event as payload of a message or composed by the event monitor is provided by the XSD-based definition in Listing 2. The initial four fields identify the event at the receiver (fist two) and sender (last two), if arriving from the environment. The tuple `sequence number` and `time` uniquely identify an event at both sides. This also supports examinations on the events actuality. The following fields `origin`, `type`, `extended type`, and `description` are mandatory. `Origin` indicates the source of the event. These include environment or composed type. The `extended type` field tags the events nature. Tags reflect hardware and communication faults, human related workload and delegation problems, and QoS and agreements related issues. The `description` field contains a human readable description of the event. This is included for offline evaluation and or online test runs assisted by humans. The final required field of the schema is the event's severity. The `severity` defines the events queuing priority and processing urgency.

Event triggers are implemented using JBoss Drools[4] to detect negative behavior of nodes. Multiple rules are defined to trigger behavior that potentially leads to unhealthy problems, such as factory or sink behavior discussed before. Listing 3 shows an excerpt of rule definitions to detect sink behavior. In particular, if a node's task queue is considerably filled (`numTasksQueued`) but does not (or nearly not) delegate to neighbors (`delegationRate`), sink behavior is detected. VieCure attempts to heal such situations by creating recovery actions (`RecoveryAction`) that lead to the insertion of additional edges, i.e., delegation channels, in the network.

```
rule "TriggerFactoryBehavior"
        when
                node:Node(delegationRate > 50 && role == "worker")
                recoveryActionList:ArrayList()
        then
                Node neighbor = Utils.lookupNodeSimilarCapabilities(node)
                RecoveryAction ctlCapacity = new CtlCapacity(neighbor, node);
                recoveryActionList.add(ctlCapacity);
end
rule "TriggerUnusualDelegationRateWorker"
        when
                node:Node(numTasksQueued > 15 && delegationRate < 2)
        then ...
end
```

Listing 3. Triggering events and setting recovery actions.

The final step in the healing process is to execute recovery actions. Listing 4 shows an example how such recovery actions can be performed in our system.

As mentioned previously, an approach for recovering from degraded system state is regulation of delegation behavior between actors (HPSs). This is accomplished by sending the corresponding recovery action to an *Activity Management Service* (see [9] for details). In Listing 4, a `ControlAction` of type `Coordination` is depicted regulating the flow of delegations between two actors. Each `Coordination` action has a unique identifier and is applied in the context of an activity. The `ControlAction` also contains what kind of `ActionType` has to be regulated as a result of a recovery. In this example regulation applies to `Delegation` actions by changing the capacity of delegation channels.

```
<ControlAction xmlns="http://myhps.org/Action"
 xmlns:vietypes="http://www.infosys.tuwien.ac.at/Type"
 xsi:type="Coordination"
 URI="Coordination#10"
 Activity="Activity#42">
  <From>http://www.expertweb.org/Actor#Harald</From>
  <ActionType>http://myhps.org/Action/Delegation</ActionType>
  <To>http://www.expertweb.org/Actor#Florian</To>
  <vietypes:ctlCapacity capacity=.../>
</ControlAction>
```

Listing 4. Control action to recover from degraded system state.

## V. Regulation of Behavior

In our self-healing algorithm for Mixed Systems we opted for a regulation of a node's behavior in a non-intrusive manner. Instead of healing misbehavior directly at the nodes, we influence their behavior by restricting delegations, establishing new delegation channels, and by redirecting work. Next, we outline the modules of our self-healing mechanism in Algorithm 1 and detail and analyze the concepts with respect to the failure scenario in Figure 5.

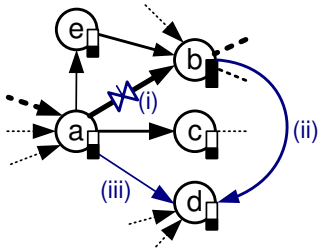



Figure 5. Self-healing recovery actions for a failure affected node.

**Trigger.** The first module (line 1 to 5), a trigger, represents a filter for the failure scenario in Figure 5. As a prerequisite any agreements and constrains monitored by this self-healing approach need to be expressed as threshold values. These values are integral part of the decision logic of a trigger module.

**Diagnosis.** A recognized violation fires the second module (line 6 to 23), the diagnosis. It defines the necessary recovery actions by analyzing the result of the task history evaluation of the failing node.

**Recovery Actions.** The possible resulting recovery actions are listed in the last three modules (line 24 to 37). The first balances load of a failing node by restricting incoming delegations. The second provides the failing node with new delegation channels for blocked tasks. The last

[4]http://jboss.org/drools

**Algorithm 1** Detection of misbehavior and recovery actions.

**Require:** Monitoring of all nodes
**Require:** Listen to Events
1: **Trigger** `triggerQueueOverload`(*event*)
2: $node \leftarrow event.node$ /*affected node*/
3: **if** $q > \vartheta_q$ **then**
4: fire `diagnoseBehavior`(*node*)
5: **end if**
6: /*diagnose sink and factory behavior*/
7: **Diagnosis** `diagnoseBehavior`(*node*)
8: $recActs \leftarrow \emptyset$ /*set of returned recovery actions*/
9: $recActs$.add(`addChannel`(*node*))
10: analyzeTaskHistory(*node*)
11: **for** $neighbor \leftarrow$ affectedNeighbors(*node*)
12: **if** (rankTasks(*node*) $> \vartheta_{pref}$) **or** ($p < \vartheta_p$) **then**
13: /*root cause: sink behavior*/
14: $recActs$.add(`redDeleg`(*neighbor*))
15: $recActs$.add(`ctlCapacity`(*neighbor*, *node*))
16: **else if** ($q < \vartheta_q$) **and** ($d > \vartheta_d$) **then**
17: /*root cause: factory behavior*/
18: $recActs$.add(`ctlCapacity`(*neighbor*, *node*))
19: **else**
20: /*root cause: transient degradation*/
21: $recActs$.add(`redDeleg`(*neighbor*))
22: **end if**
23: **return** $recActs$
24: /*recovery action: control capacity*/
25: **Recovery Action** `ctlCapacity`(*neighbor*, *node*)
26: *cap* $\leftarrow$ estimateCapacity(*neighbor*, *node*)
27: setCapacity(*cap*)
28: /*recovery action: add channel*/
29: **Recovery Action** `addChannel`(*node*)
30: $simNode \leftarrow$ lookupNodeSameCapabilities(*node*)
31: addDelChannel(*node*, *simNode*)
32: `ctlCapacity`(*node*, *simNode*)
33: /*recovery action: redirect delegations*/
34: **Recovery Action** `redDeleg`(*neighbor*)
35: $simNode \leftarrow$ lookupNodeRequiredCapabilites(*neighbor*)
36: addDelChannel(*neighbor*, *simNode*)
37: `ctlCapacity`(*neighbor*, *simNode*)

assists neighbors by providing new delegation channels to alternative nodes.

As mentioned before, a loop-style data-flow between the guarded system and the self-healing mechanism allows to observe changes. Changes leading to possible failures are recognized by the mechanism by directing the data-flow through the trigger modules' logic. In Algorithm 1 Trigger `triggerQueueOverload` filters events which indicate a threshold violation of the task queue capacity of a node (Line 3). Such an event causes `triggerQueueOverload` to fire the related diagnosis `diagnoseBehavior` passing on the failure affected *node* information. E.g., in Figure 5 the congestion of node $b$ is reported as such an event.

As a first precaution in `diagnoseBehavior` the algorithm balances the load at *node* and adds recovery action `addChannel` to the recovery result-set $recActs$. The idea is to relieve *node* by providing *node* with new delegation options to nodes with sufficiently free capacities. The task of this recovery action is to discover a node that has capabilities similar to *node*. Once the delegation channel is added, in `ctlCapacity` method $estimateCapacity$ estimates the maximum possible of task transfer regarding the discovered nodes' processing capabilities. Finally, $setCapacity$ controls the throughput accordingly. Next, in $analyzeTaskHistory$ the diagnosis derives a root cause from the reported node's task history. A repository of classified failure patterns is compared to the last behavior patterns of the node and the corresponding root cause returned. In a loop (line 11), by selecting the affected neighbors, behavior is analyzed.

**Sink Behavior.** Line 12 identifies sink behavior. The result of the pattern analysis shows that *node* is still accepting tasks from any neighbor, however, prefers to work on tasks of a certain neighbor and delays the tasks of the other nodes. The second misbehavior of a sink is to perform tasks below an expected rate ($p < \vartheta_p$). The additional counter actions try to provide options for the set of affected delegating neighbor nodes and to decouple the sink. Recovery action `redDeleg` finds the alternatives and again estimates the adequate capacity of the new delegation channels. Recovery action `ctlCapacity` sets the delegation rate between sink and its neighbors to a minimum. The situation is depicted in Figure 5. Delegation channel (ii) is added from $b$ to similar capable node $d$ and allows $b$ to dispense a certain amount of capability matching tasks. Delegation channel (iii) from $a$ to $d$ is a result of `redDeleg`. In our example, $d$ has enough resources to process blocked (from $b$) and new tasks (from $a$). The amount of recently delegated tasks is balanced in `estimateCapacity`. Thereafter the capacity of delegation channel (i) is minimized. A limitation of the delegations depends on the content of $b$'s task queue. The example assumes that it mostly contains tasks from $a$. If the capacity of delegation channel (iii) is too low for $a$'s delegation requirements, it might consider to process the tasks itself, or discover an additional node for delegation. The whole scenario is also applicable for a factory behavior of $a$. In that case, further uncontrolled delegations of $a$ are avoided and no new delegation channel (iii) would be added.

**Factory Behavior.** Line 16 detects a delegation factory behavior. A factory is identified by moderate use of queue capacity ($q < \vartheta_q$) in contrast to high and exceeding delegation rates ($d > \vartheta_d$) causing overloaded nodes despite available alternatives. Recovery restricts the delegations from the

factories to *node*, expecting that the factories start increasing their task processing performance or find themselves other nodes for delegations. Besides releasing the load from *node*, `ctlCapacity` ensures that the delegation of tasks from a factory to *node* is set to a minimum.

**Transient Behavior.** In Line 19, if neither factory nor sink behavior are recognized `diagnoseBehavior` must assume a temporal overload of *node*. As a second precaution the algorithm estimates alternative delegation nodes in `redDeleg` for the neighbors of *node*.

## VI. Simulation and Evaluation

In our experiments we evaluate the effectiveness of previously presented recovery action algorithms (c.f., Section V) in a simulated mixed SOA environment. Figure 6 outlines the controllable simulation environment on the left used for our experiments. We took interaction logs from the real mixed SOA environment on the right to reconstruct the main characteristics.

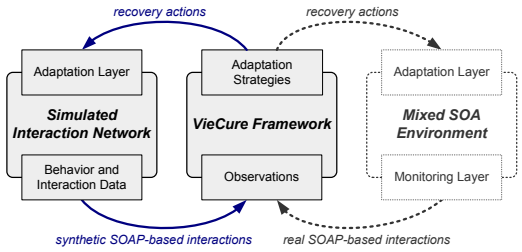



Figure 6. Simulation setup.

### A. Simulation Setup

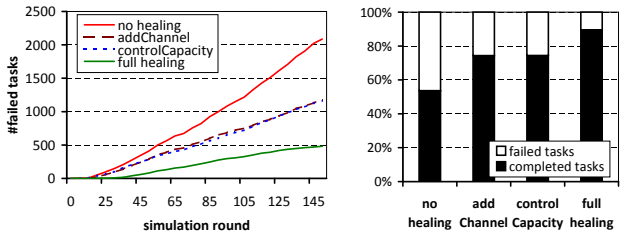
**Simulated Heterogeneous Service Environment.** The simulated interaction network comprises a node actor framework implemented in JAVA language. At bootstrapping the nodes receive a profile including different behavior models. Each node has a task list with limited capacity. Depending on the deployed behavior model a node tends either to delegate, or process tasks, or exposes a balanced behavior. New tasks are constantly provided to a quarter of the nodes via connected entry points. Tasks have an effort of three units. A global timer initiates the simulation rounds. Depending on the behavior model, in each round a node decides to process tasks or delegate one task. A node is able to process the effort of a whole task, or if delegating, only one effort unit. For the delegation activity a node holds a current neighbor list which is ordered according to the neighbors' task processing tendency. The delegating node prefers nodes with processing behavior and assigns the selected the longest remaining task. A receiving node with a task queue at its upper boundary refuses additional tasks. However, each task is limited by a ten round expiry. If a task is not processed entirely in this period it is considered a *failed task*.

**VieCure Setup.** At bootstrapping the VieCure monitoring and adaptation layer is instantiated. In our simulated environment the monitor has an overview over all nodes. Thus, the monitor provides the VieCure framework with a current node list together with their task queue levels. A trigger filters the queues' levels and reports to diagnosis if the lower threshold value is exceeded. Diagnosis estimates then the actual level and decides on the recorded history together with the current situation which recovery action to choose. For the purpose of the evaluation of the recovery actions, we required diagnosis to act predictable and decide according to our configuration which recovery action to select.

**Recovery actions** Two of the outlined recovery actions in Section V were implemented. In *control capacity*, the delegation throughput to the affected node is adapted according to the current task queue level. In *add channel*, the filtered node is provided with a new channel to the node with the currently lowest task queue load factor. In order to evaluate the effects of the recovery actions we executed four different runs with the same setting. At the end of each experiment the logging facilities of the VieCure framework provided us with all the information needed for analysis. The results are presented next.

### B. Results and Discussion

The experiments measure the efficiency of a recovery action by the amount of *failed tasks*. An experiment consists of a total number of 150 rounds and a simulation environment with 128 nodes. During an experiment 4736 tasks are assigned to the nodes' network. In order to prevent an initial overload of a single node as a result of too many neighbor relations, we limited the amount of incoming delegations channels to a maximum of 6 incoming connections at start-up. The resulting figures present on their left the total of failed tasks after a certain simulation round. The curves show the progress of different configurations of VieCure's diagnosis module. The figures on the right represent the ratio failed/processed tasks in percentages at the end of the experiments with an equal setting.

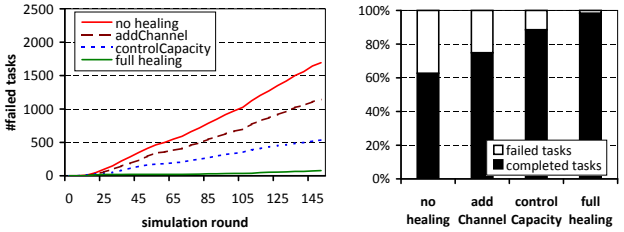



(a) Current failure rate. (b) Final overal success rate.

Figure 7. Equal distirbution of behavior models.

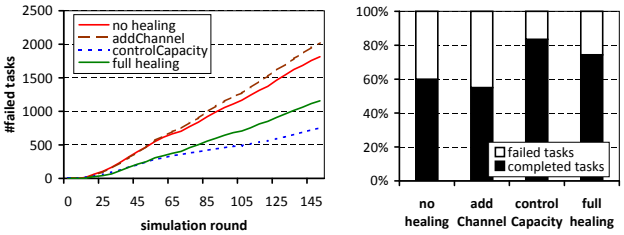
The setting for the results in Figure 7 consisted of an equal number of the three behavior models distributed among the nodes. Whilst the nodes on their own produce a total of 2083 failed tasks (top continuous curve) the two different recovery actions separately expose an almost equal progress and finish at almost half as much; 1171 for *add channel* action and

1164 for *control capacity* action, respectively. Combining both diminishes the failure rate to a quarter compared to no action, to 482 failed tasks (lower continuous curve). The results demonstrate that in an equilibrated environment our two recovery actions perform almost equal and complete each-other when combined.




(a) Current failure rate. (b) Final overall success rate.

Figure 8. Distribution with a trend for 10% factory behavior.




(a) Current failure rate. (b) Final overal success rate.

Figure 9. Distribution with a trend for 10% sink behavior.

In Figure 8 the setting configured a tenth of nodes with factory tendency and an equal distribution of the other two models across the remaining nodes. An immediate result of the dominance of task processing nodes is that less tasks fail generally. The failure rate for the experiment with no recovery falls to a total of 1693 (top continuous curve). The success of *add channel* (dashed curve) remains almost the same (1143). With this unbalanced setting the potential neighbors for a channel addition remain, however, the same as in the previous setting. In contrast, the success of *control capacity* (dotted curve, 535) relies on the fact that regulating channels assures that the number of tasks in a queue relates to the task processing capabilities given by a node's behavior. In strategy combination (lower continuous curve, 77), this balancing mechanism is supported by additional channels to eventually still failing nodes. The results are also reflected by the success rate figure. In Figure 9 the setting was changed to a 10% of sink behavior trend. Without a recovery strategy the environment performs almost the same as in the previous setting (top continuous curve, 1815). The strategy of just adding channels to overloaded nodes fails. Instead of relieving nodes from the task load, tasks circle until they expire. Thus, a number of 2022 tasks fail for *add channel* (dashed curve). The figure further shows, that this problem has also impact on the combination of the two strategies (lower continuous curve, 1157). The best solution for the setting is to inhibit the dominating factory behavior by controlling the channels capacity (dotted curve, 753).

## VII. Related Work

The concepts of self-healing are applicable in various research domains [4]. Thus, there is a vast amount of research available on self-healing designs for different areas. These include higher layers such as models and systems' architecture [10], [11] application layer, and in particular interesting for our research are large-scale agent-based systems [12], [13], [14], Web services [15] and their orchestration [16]. In the middle, self-healing ideas can be found for middleware [17], [18], and at a lower layer self-healing designs include operating systems [19], [20], embedded systems, networks, and hardware [21]. The two main emerging directions that include self-healing research are provided by autonomic computing [7],[22] and self-adaptive systems [3]. Whilst autonomic computing includes research on all possible layers, self-adaptive systems focus primarily on research above the middleware layer with a more general approach.

With current systems growing in size and ever changing requirements plenty of challenges remain to be faced such as autonomic adaptations [6] and service behavior modeling [23]. The self-healing research demonstrated in this paper relates strongly to the challenges in Web services and workflow systems. Apart from the cited, substantial research on self-healing 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-healing strategies and adaptation of BPEL processes [24], [15]. Others are theoretical discussions on self-healing methodologies [25].

Human-Provided Services [2] 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, 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-healing method recovers by soundly restricting delegation paths or establishing new connections between the nodes.

## VIII. Conclusion and Outlook

In our work we analyze misbehavior in Mixed Systems with our novel VieCure framework comprising an assemble of cooperating self-healing modules. We extract the monitored misbehaviors to models and diagnose them with our self-healing algorithms. The recovery actions of the

algorithm heal the identified misbehaviors in non-intrusive manner. The evaluations in this work shown that our elaborate recovery actions compensate satisfactorily the misbehaviors in a Mixed System (about 30% higher success rate with equal distribution of behavior models). The success rates of the recovery actions depend on the environment settings. In all but one of the cases, deploying recovery actions supports the overloaded nodes resulting in a higher task processing rate. Important to note, that the failure rate increase near linearly even when recovery actions adjust the nodes' network structure. This observation emphasizes our attempt in implementing non-intrusive self-healing recovery strategies.

Future work will involve the integration of VieCure into the GENESIS testbed framework [26] in order to interface the controlling capabilities of the framework with VieCure's self-healing implementations. Experiments in this testbed environment will provides us with more accurate data when extending VieCure with additional self-healing policies to cover new models of Mixed System's misbehavior.

## ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (SCube) and 216256 (COIN).

## REFERENCES

[1] 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.

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

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

[4] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya, "Self-healing systems - survey and synthesis," *Decis. Support Syst.*, vol. 42, no. 4, pp. 2164–2185, 2007.

[5] H. Psaier and S. Dustdar, "A survey on self-healing systems - approaches and systems," *Computing*, vol. 87, no. 1, 2010.

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

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

[8] F. Skopik, D. Schall, and S. Dustdar, "Trusted interaction patterns in large-scale enterprise service networks," in *Euromicro PDP*, 2010, pp. 367–374.

[9] D. Schall, C. Dorn, S. Dustdar, and I. Dadduzio, "Viecar - enabling self-adaptive collaboration services," in *SEAA '08*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 285–292.

[10] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *WOSS*, 2002, pp. 21–26.

[11] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. P. Sousa, B. Spitnagel, and P. Steenkiste, "Using architectural style as a basis for system self-repair," in *WICSA*, 2002, pp. 45–59.

[12] S. Corsava and V. Getov, "Intelligent architecture for automatic resource allocation in computer clusters," in *IPDPS*, 2003, p. 201.1.

[13] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing," in *AAMAS*, 2004, pp. 464–471.

[14] 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 Systems Journal*, vol. 41, no. 3, pp. 350–371, 2002.

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

[16] L. Baresi, S. Guinea, and L. Pasquale, "Self-healing bpel processes with dynamo and the jboss rule engine," in *ESSPE*, 2007, pp. 11–20.

[17] 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.

[18] T. Ledoux, "Opencorba: A reflektive open broker," in *Reflection*, 1999, pp. 197–214.

[19] A. Tanenbaum, J. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, vol. 39, no. 5, pp. 44–51, 2006.

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

[21] M. Glass, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich, "Symbolic reliability analysis of self-healing networked embedded systems," in *SAFECOMP*, 2008, pp. 139–152.

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

[23] K. Kaschner and K. Wolf, "Set algebra for service behavior: Applications and constructions," in *BPM '09*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 193–210.

[24] 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.

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

[26] L. Juszczyk, H.-L. Truong, and S. Dustdar, "Genesis - a framework for automatic generation and steering of testbeds of complexweb services," in *ICECCS'08*, 2008, pp. 131–140.