A Programming Model for Self-Adaptive Open Enterprise Systems

Harald Psaier, Florian Skopik, Daniel Schall
Lukasz Juszczyk, Martin Treiber, Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstraße 8/184-1, A-1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

ABSTRACT

Open Web-based and social platforms dramatically influenced models for work. The emergence of service-oriented systems has paved the way for a new computing paradigm that not only applies to software services but also human actors. This work introduces a novel programming model for *Open Enterprise Systems* whose interactions are governed by dynamics. Compositions of humans and services often expose unexpected behavior because of sudden changes in load conditions or unresolved dependencies. We present a middleware for programming and adapting complex service-oriented systems. Our approach is based on monitoring and real-time intervention to regulate interactions based on behavior policies. A further challenge addressed by our approach is how to simulate and adapt behavior rules prior to deploy polices in the real system. We outline a testing approach to analyze and evaluate the behavior of services.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; H.3.5 [**Online Information Services**]: Web-based Services; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Design, Human Factors, Management, Performance

Keywords

Open Enterprise, Programming Model, Human Dynamics

1. INTRODUCTION

Service-oriented architecture (SOA) is an emerging paradigm to realize extensible large-scale systems. As interactions and compositions spanning multiple enterprises become increasingly commonplace, organizational boundaries appear to be diminishing in future service-oriented systems.

In such open and flexible enterprise environments, people contribute their capabilities in a service-oriented manner. We consider service-oriented systems based on two elementary building blocks: (i) software-based services (SBS), which are fully automated services and (ii) human-provided services (HPS) [13] for interfacing with people in a flexible service-oriented manner. Related efforts in service-oriented systems such as WS-HumanTask [10] attempt to model human interactions in top-down business processes assuming closed enterprise systems.

Here we discuss service-oriented environments wherein services can be added at any point in time. Following the open world assumption, humans actively shape the availability of HPSs. Without any coordination, such systems may exhibit undesirable properties due to unexpected behavior. Thus, social implications caused by human participation pose additional challenges to designing large-scale mixed SBS-HPS systems. However, with size also the effort for managing dynamically growing and loosely coupled systems is sharply increasing. Periodic adaptations are essential to keep a system within well-defined states, including stable load conditions or desired behavior. Due to the scale and inherent dynamics of open large-scale systems, new approaches for designing, developing and testing are required.

1.1 Motivating Scenario

Let us start with a service-oriented collaboration scenario that motivates our work. 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 dynamically 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.

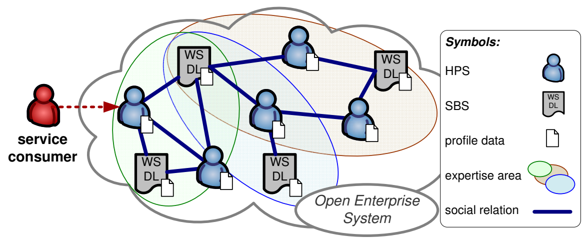


Figure 1: Mixed Open Enterprise System.

We model a mixed Open Enterprise System (OES) consisting of HPSs [13] and SBSs that belong to different communities. The members of these communities are discovered based on their capabilities (see profiles) and main expertise areas (depicted as shaded areas), and are connected through certain relations (e.g., FOAF[1]). 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 *OES*). 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 loose coupling and flexible involvements of human experts in a service-oriented manner. 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.

Failures and performance degradations may arise due to various reasons when operating an OES. For instance, performance degradations can be expected when a minority of distinguished experts become flooded with tasks while the majority remains idle. Such load distribution problems can be compensated by adapting the *delegation behavior* of actors as discussed in [14]. Furthermore, consider shifting interests and evolving skills of people. In that cases capabilities of HPSs as well as people's interaction behavior may change over time, thus, requiring again adaptations of the underlying infrastructure.

1.2 Approach and Contributions

Our approach to ensure the smooth operation of mixed OESs makes use of several modules. The main contribution of this paper is the description of these modules, their modes of operation and discussions on major design decisions. We highlight the contributions of this work in Figure 2:

- *Open Enterprise System* hosts both SBSs and HPSs interacting to perform joint activities.
- *VieCure Middleware* provides a programming model including monitoring and adaptation mechanisms that control the OES *and* the simulation environment. Service policies to regulate behavior (interaction dynamics) are based on observations and control.

  Adaptation strategies can be deployed in the Genesis Simulation Environment for testing purposes. Furthermore, logs, from either the simulation or the life OES can be analyzed to customize configurations and adaptation strategies.
- *Genesis Simulation Environment* allows to investigate the effects of policies and adaptation strategies.

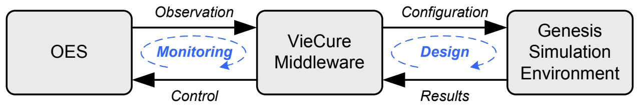


Figure 2: Approach outline and contributions.

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

Paper Outline. The remainder of this paper is structured as follows. Section 2 details our flexible adaptation approach and the prototype implementation of a novel middleware. Furthermore, we highlight the configuration management and the Genesis2 testbed generator framework. A discussion on collected experience and findings are provided in Section 3. We outline related work in Section 4 and conclude the paper in Section 5.

2. DESIGN AND ARCHITECTURE

In this section we introduce our programming model for adaptations in OES. After outlining the *VieCure* middleware's components, we provide a sample environment specification for a OES in Genesis2's programming language. This helps to highlight how our newly introduced programming model maps high-level policies to actions deployable to the environment. We conclude by outlining how different environments can be modeled, simulated, hosted, controlled and, of course, adapted with the programming model.

2.1 Middleware for Adaptation

At its core the middleware for adaptation provides a programming model for policy rules and actions which transparently adapt the OES underneath. The interface to the *Configuration Manager* offers monitoring and visualization features for the current OES structure. Additionally, it allows to adjust the middleware's policies with rules and actions from a more high-level and goal-driven perspective.

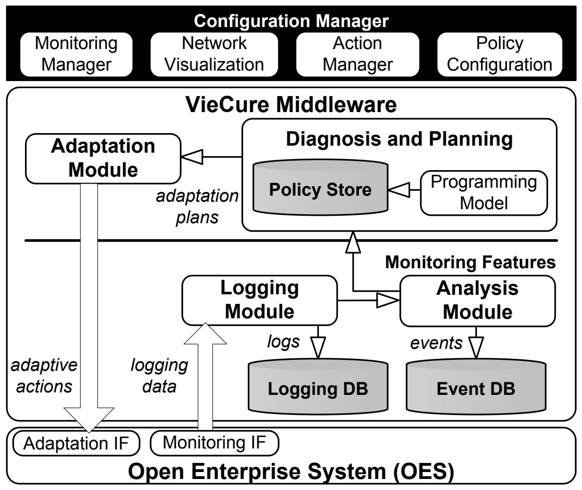


Figure 3: Middleware for adaptive OESs.

The three-layer infrastructure of the *VieCure Middleware* is outlined in Figure 3. The top layer comprises the *Configuration Manager*. This is a tool-set to monitor, track, and analyze the OES structure. Additionally, by hiding the particularities of the programming model it allows to extend and change the entries of the middleware's *Policy Store* easily and, thus, to influence the following adaptations.

The layer in the middle hosts the *VieCure Middleware*. It is organized according to a MAPE-K loop [6], the common design pattern used for self-adaptive systems. Therefore, it connects and forwards event information through the modules monitoring, analyzing, planing, and adaptation in loop-style. Different databases assist the analyses including evaluation with event history. The loop connects the *Logging Module* to the *Monitoring Interfaces*. The data is converted to processable events in the *Analysis Module*. Events activate an examination of the rules in the *Diagnosis and Planning* module. The actions related to the triggered rules are then deployed to the OES by the *Adaptation Module*. All policies are defined by our *Programming Model* which provides a language specification to define the policy's rules together with the actions. The following sections explain the concepts of the middleware and the surrounding layers in more detail.

2.2 Genesis WS Framework

The purpose of the Genesis2 framework (in short, G2) is to provide all means to host a SOA based environment including OESs. Originally designed to model Web services testbeds, we discuss G2 in the context of modeling, programming, and adapting OES.

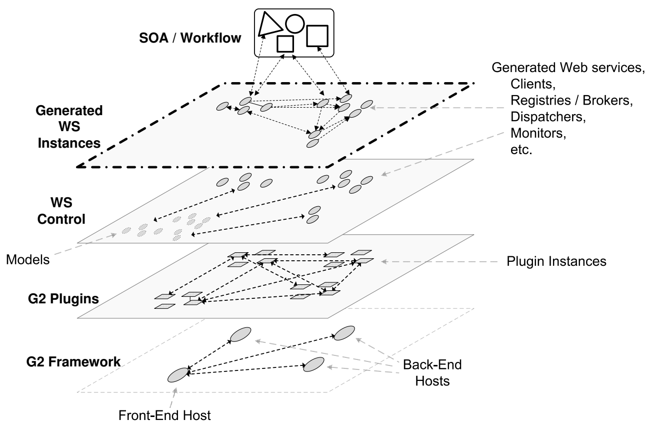


Figure 4: Layered G2 topology.

Figure 4 shows G2's layered topology consisting of various elements. This layered topology enables software engineers to generate and control SOA environments. The top layer displays the runtime with the *Generated WS Instances* including services, clients for service, registries, etc. The next layer enables control over the generated instances. The *WS Control* layer comprises a comprehensive model of the deployed environment. It allows to steer the execution of the instances and propagates any model changes back to the environment. The following *G2 Plugins* layer allows to dynamically contribute external extensions. These could be environment specific and include, e.g., studied behavior models that are tested by the currently deployed environment.

G2 provides its own programming model and language. This is an extension of the Groovy[2] script language with additional keywords and structures integrating the model to the language. Listing 1 gives a few examples. The model `datatype` allows to import an external complex data type with the method *create()*. The statement in line 3 shows the use of the `webservice` model to define an array of WS bodies with different properties and operations. Lines 25 and 26 demonstrate how a host is set-up with the `host` model and then deployed with the `webservice` model's *deployAt()* method. We will also refer to these models in our policy programming model presented next when we address the

[2]`http://groovy.codehaus.org/`

```
1  def repType=datatype.create("file.xsd","typeName") // xsd import
2
3  def srv=webservice.build {
4    // create web service
5    Service(binding:"doc,lit", namespace:"http://...") {
6      def reportQueue = [] //queue of reports
7      sendResult(input:repType, resonse:int) { //
8          def repId = genReportId()
9          if (dStrat(input))
10           reportQueue+=input
11         return repId
12     }
13     // result acceptance strategy as closure variable
14     dStrat={ input −> ...}
15     getStatus(repId:int, response:String) {
16         return getStatusOn(repId)
17     }
18     getReport(repId:int, response:String) {
19         if (completed(repId))
20           return getLocation(repId)
21     }
22   }
23 }[0]
24
25 def h=host.create("somehost:8181") // import back−end host
26 srv.deployAt(h) // deploy service at remote back−end host
27
28 dStrat={ input −> ...} // change strategy at runtime
```

Listing 1: Sample WS environment specification.

formulation of adaptive actions. For a more detailed description of G2 and its model we refer readers to [7, 16].

2.3 Programming Model for Policies

The *VieCure* middleware provides its own programming model for polices. Policies comprise rules and connected actions. In line with the event, condition, action paradigm, once an event triggers a rule because it matches the condition, e.g., threshold of a metric, an action is executed and an adaptation deployed to the environment. An event passed from the *Analysis Module* or manual intervention from the *Configuration Manager* activates rule evaluation.

In order to design a policy rule, we use JBoss Drools[3]. The structure of such a list of rules is presented in Listing 2. It illustrates three example rules matching the scenario of behavior adaptation outlined in Section 1.1. All rules are defined by a starting `rule` tag. The following `when` states the condition on which the rule fires. The `then` part defines the action that is deployed. By providing an array list multiple actions can be add and deployed simultaneously depending on the event type. In the example the first rule `Overload` deploys a suspend operation action if the workload or traffic to a specific service becomes intolerable. The next rule `Deprecated` removes an operation if several previous invocation trials fail. Finally, the third rule `Add` extents the functionality of a service by adding a new operation. The last rule can be considered one that was added through the *Configuration Management*. In the present case the system designer, e.g., decided to extend a service with a new adaptation rule because new capabilities can be offered. The rule fires in any case and, thus, has an empty condition.

It is important to note, that all of the recovery actions in Listing 2 take a parameter `c` as input. This parameter represents a placeholder for a policy action definition. The

[3]`http://jboss.org/drools`

```
rule "Overload"
    when    //too many messages to process
        service:Service(receiveRate > 50 || workload > 0.75)
        adaptationActionList:ArrayList()
    then
        AdaptationAction ra = new SuspendOpAction(c)
        adaptationActionList.add(ra);
end
rule "Deprecated"
    when    //operation is not available any more
        service:Service(invokeRetry > 10)
        adaptationActionList:ArrayList()
    then
        AdaptationAction ra = new RemoveOpAction(c)
        adaptationActionList.add(ra);
end
rule "Add"
    when    //add new operation
        adaptationActionList:ArrayList()
    then
        AdaptationAction ra = new AddOpAction(c)
        adaptationActionList.add(ra);
end
```

Listing 2: Policy rules with recovery actions.

```
1   //--- change a operation
2   webservice(name:srvname) {s -> name in s.name} { s->
3     if ("sendResult" in s.operations.name) {
4       def op = s.operations.grep{o -> o.name == "sendResult"}[0]
5       op.behavior = {
6         return "Operation temporarily suspended"
7       }
8       op.returnType=String
9       s.redeploy()
10    }
11  }
12  //--- ws operation removal
13  webservice(name:srvname) { s-> name in s.name} { s->
14     def o = s.getOperation("sendResult") //get operation
15     s.operations-=o //delete operation
16     s.redeploy() //redeploy service and wsdl
17  }
18  //--- add a new operation
19  webservice(name:srvname) { s-> name in s.name}{ s->
20    s.operations += wsoperation.build {
21      delegateResults(input:repType) {
22        delegate(input)
23    }
24   }[0]
25   s.redeploy()
26  }
```

Listing 3: Remove an expired service operation.

definition depends on the adaptation interface requirements of the hosting environment. To give an insight on how adaptive actions can be created we provide in the following some samples that are executable on G2 hosts. G2 requires the parameter `c` to represent actions defined in G2's Groovy-based programming language [7]. As pointed out, G2 provides model modification via the control layer. Changes on the model are automatically propagated to the running environment.

Listing 3 shows three example closures (code blocks[4]) matching the input parameters of the previously stated adaptation rules in Listing 2. All closures presented access the `webservice` model introduced briefly earlier. Apart from defining Web services this model is also used to change the deployed instances of services at runtime. In the following cases the `webservice` model takes a input variable of any known type that can be used to filter particular service instances. Thus, the first closure definition of the model works as a filter for the affected instances. In the example cases the service definitions are selected by the variable *name*, e.g., provided by the triggering event. After changes are applied to the model, changes on the service models are deployed to the running environment at the end (method *redeploy()*).

In detail, the meaning of the adaptation actions stated in Listing 3 is the following. Rule `Overload` requires a temporary suspension of an operation. In the corresponding action we assume that diagnosis recognized that operation *sendResult()* (c.f., Listing 1) causes overloads to the service because it fills the result document queue. The action definition from line 4-9 in Listing 3 states that the affected services' operation is overwritten by a simple code that returns a suspension message. For the next rule `Deprecated` the same operation is taken offline. The related action code shows in line 14 another `webservice` model method to get a distinct operation of the service model and in line 15 a method to remove the deprecated method.

Finally, for the `Add` rule in line 20 we require another model of the G2 programming model. The same as with `webservice`, `wsoperation` allows to create an operation by defining it in the closures content. After the change, the resulting service additionally to process result documents is capable to forward the documents by delegation.

The reader is reminded that any considerations on the impacts of the adaptations on the implemented clients are out of scope. Nevertheless, the *redeploy()* calls assure that also the new service definition is deployed and clients would be able to adjust to the new interface definitions.

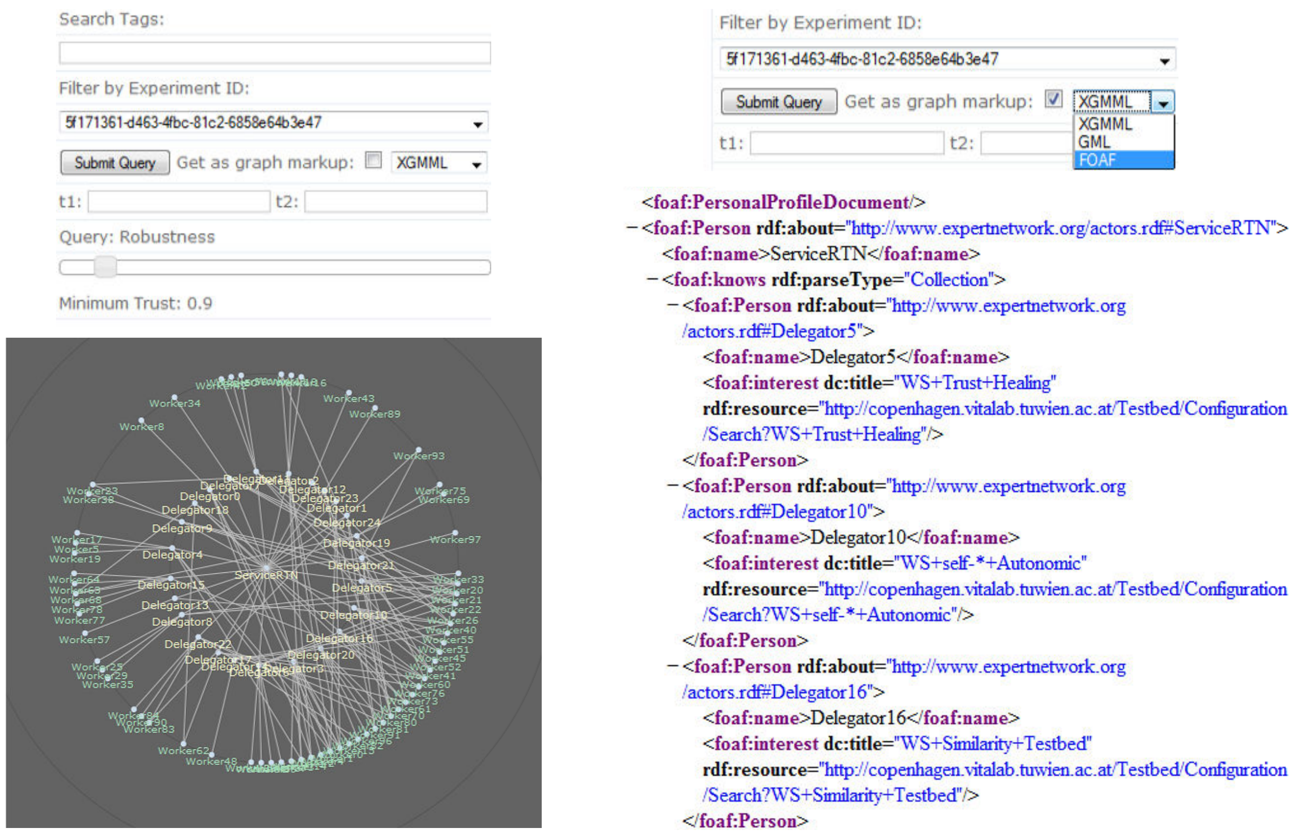
2.4 Simulation of Human Behavior

One of the key questions in social dynamics concerns the behavior of single individuals, namely how an individual chooses a convention, takes a decision, schedules his tasks and more generally decides to perform a given action. Most of these questions are obviously very difficult to address, due to the psychological and social factors involved [5]. This problems are also of particular interest in collaborative service environments, as their interaction patterns and usage dependent on human behavior. Instead of focusing on the individual's behavior in cooperative systems only (e.g., as discussed in [1]), we focus on the *network effects* of human dynamics.

A unique feature of the presented G2 service hosting environment is its capability to not only be used as a hosting environment but also to run simulations of service environments as hinted in Figure 2 in the introduction. Simulations can be used to evaluate the effectiveness of, e.g., the previously presented and defined policies, on varying environment conditions. As the motivation explains, we are in particular interested in service environments including human actors. In such simulations human behavior including making choices, taking decisions, working on tasks, or performing actions, need to be simulated with the help of previously observed and examined log data.

The G2 programming model introduced in Section 2.2 ap-

[4]`http://groovy.codehaus.org/Closures`


Search Tags:

Filter by Experiment ID: 5f171361-d463-4fbc-81c2-6858e64b3e47

Submit Query Get as graph markup: ☐ XGMML

t1: t2:

Query: Robustness

Minimum Trust: 0.9

Filter by Experiment ID: 5f171361-d463-4fbc-81c2-6858e64b3e47

Submit Query Get as graph markup: ☑ XGMML
XGMML
GML
FOAF

t1: t2:

```
<foaf:PersonalProfileDocument/>
– <foaf:Person rdf:about="http://www.expertnetwork.org/actors.rdf#ServiceRTN">
    <foaf:name>ServiceRTN</foaf:name>
  – <foaf:knows rdf:parseType="Collection">
    – <foaf:Person rdf:about="http://www.expertnetwork.org
      /actors.rdf#Delegator5">
        <foaf:name>Delegator5</foaf:name>
        <foaf:interest dc:title="WS+Trust+Healing"
        rdf:resource="http://copenhagen.vitalab.tuwien.ac.at/Testbed/Configuration
        /Search?WS+Trust+Healing"/>
      </foaf:Person>
    – <foaf:Person rdf:about="http://www.expertnetwork.org
      /actors.rdf#Delegator10">
        <foaf:name>Delegator10</foaf:name>
        <foaf:interest dc:title="WS+self-*+Autonomic"
        rdf:resource="http://copenhagen.vitalab.tuwien.ac.at/Testbed/Configuration
        /Search?WS+self-*+Autonomic"/>
      </foaf:Person>
    – <foaf:Person rdf:about="http://www.expertnetwork.org
      /actors.rdf#Delegator16">
        <foaf:name>Delegator16</foaf:name>
        <foaf:interest dc:title="WS+Similarity+Testbed"
        rdf:resource="http://copenhagen.vitalab.tuwien.ac.at/Testbed/Configuration
        /Search?WS+Similarity+Testbed"/>
      </foaf:Person>
```

(a) Network visualization view. (b) Example of (FOAF-based) network profile.

Figure 5: Web-based configuration management tool.

plies the concept of closures to equip services with individual behavior. Referring to lines 9, 14 and 28 in the sample script of Listing 1 it can be recognized that *dStrat* is defined as a global closure in the service model. While in the example the running service expects an executable algorithm resulting in an accept or reject of a received document the strategy can be changed at any time during runtime by setting a new closure content to the global placeholder (line 28). Also, with this method the strategy can be set individually for the otherwise identical service definition. This makes the simulation more authentic. Furthermore, once VieCure's adaptation loop is activated, adaptation strategies are deployed to the simulated environment and conclusions on the effectiveness of adaptation strategies can be made.

2.5 Configuration Management

Figure 5 shows screenshots of the Web-based configuration management tool and an example FOAF profile that can be retrieved from the Web application.

The purpose of the network visualization view as shown by Figure 5(a) is to analyze complex behavior in OES environments. The view is based on a graph structure modeled as $G = (V, E)$ where V represents the set of services (HPS and SBS) and E the set of edges based on interactions. The network view is obtained by mapping raw SOAP-interactions into a graph representation composed of nodes (services) and edges (interaction links). The users access information captured from the OES (Figure 5). In our implementation, this is performed by selecting a particular set of logs which are associated with an `Experiment ID`. By default, the collaboration network is visualized as a graph view as depicted in Figure 5(a). The user is able to select a particular metric (in this case trust) threshold by moving a slider bar. A reduced metric threshold results in more target nodes and edges being added to the visualization. Interactions can be retrieved as FOAF profiles (see Figure 5(b)) that include `<foaf:interest>` tags. FOAF-based network profiles are especially useful for analyzing relations using semantic reasoning engines. Also RDF-based query languages such as SPARQL[5] can be used to query network data. This mechanism can be used to retrieve and aggregate captured profiles from distributed environments (e.g., from multiple instances of the logging service).

3. DISCUSSION AND FINDINGS

OESs pose a number of new challenges with their flexibility and their tendency to unexpected behavior. Todays SOA infrastructures can provide the necessary flexibility, with easily operable interfaces and interaction channels enabling communication and collaboration. However, those do not provide means to handle unpredictable changes or system degradations. With humans collaborating in such networks, there are usually no preplanned top-down composition models. This results in changing interaction and behavior patterns that possibly contradict and at times result in faults from varying conditions and misbehavior in the network. A conflict resolution management for such an infrastructure must not only monitor events but also react

[5]`http://www.w3.org/TR/rdf-sparql-query/`

situation-dependent according to rules. For this purpose, the VieCure middleware provides two interfaces for high flexibility. One towards the system, with the capabilities to gather log data and send adaptive actions. The other offers monitoring information and configuration handles for environment management above the middleware. In order to be consistent with self-adaptive methodologies, VieCure implements a loop-style log data transformation from event to adaptive action through the filter of the analysis modules and the policies of diagnosis. One of the major contributions of this work is the policy programming model for VieCure's *Policy Store*. The presented model allows the *Configuration Manager* to state rules that are triggered on events. Rules contain placeholder for the actions. The implementation of the actions depends on the action interface of the environment. In the presented study, actions are stated in the programming model language of the test environment. The reason is to test the created rules previous to deployment to a real environment.

4. 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 [15] and the other manifested by the research on self-adaptive systems [12]. 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. Regarding **runtime evaluation**, several approaches have been developed which could be applied for testing adaptation mechanisms. SOABench [3] and PUPPET [2], 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 [7], which allows to extend testbeds with plugins, we are able to implement scenarios which behave flexible enough to test diverse adaptation mechanisms [11]. **Human-Provided Services** [13] close the gap between SBS and humans desiring to provide their skills and expertise as a service in a collaborative process. Instead of a strict predefined process flow [9], these systems are denoted by ad-hoc contribution requests and loosely structured collaborations. The required flexibility induces even more unpredictable system properties responsible for various faults. The contributed middleware for self-adaptation and testing approach enables the recovery by restricting, for example, delegation paths or establishing new connections between services. The availability of rich and plentiful data on human interactions in **social networks** has closed an important loop [8], allowing one to model social phenomena and to use these models in the design of new computing applications such as crowdsourcing techniques [4].

5. CONCLUSION AND OUTLOOK

The main objective of this work was to introduce a middleware with adaptation features based on a novel programming model. We illustrated the adaptation of complex interactions in OESs using the programming model.

In future work we plan to evaluate the adaptation framework with various kinds of collaboration networks including recent crowdsourcing and other distributed problem-solving platforms. 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.

Acknowledgment

This work received funding from the EU FP7 program under the agreements 215483 (SCube) and 215175 (COMPAS).

6. REFERENCES

[1] W. M. Aalst. Process-aware information systems: Lessons to be learned from process mining. *Transactions on Petri Nets and Other Models of Concurrency II*, pages 1–26, 2009.

[2] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini. Model-Based Generation of Testbeds for Web Services. In *TestCom/FATES*, pages 266–282, 2008.

[3] D. Bianculli, W. Binder, and M. L. Drago. Automated Performance Assessment for Service-Oriented Middleware. Technical Report 2009/07, 2009.

[4] D. Brabham. Crowdsourcing as a model for problem solving: An introduction and cases. *Convergence*, 14(1):75, 2008.

[5] C. Castellano, S. Fortunato, and V. Loreto. Statistical physics of social dynamics. *Reviews of Modern Physics*, 81(2):591–646, May 2009.

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

[7] L. Juszczyk and S. Dustdar. Script-based generation of dynamic testbeds for soa. In *ICWS*, pages 195–202. IEEE Computer Society, 2010.

[8] J. Kleinberg. The convergence of social and technological networks. *Commun. ACM*, 51(11):66–72, 2008.

[9] F. Leymann. Workflow-Based Coordination and Cooperation in a Service World. In *CoopIS, DOA, GADA, and ODBASE*, pages 2–16, 2006.

[10] M. Amend et al. Web Services Human Task (WS-HumanTask), Version 1.0, 2007.

[11] H. Psaier, L. Juszczyk, F. Skopik, D. Schall, and S. Dustdar. Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems. In *SASO*. IEEE, 2010.

[12] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2):1–42, 2009.

[13] D. Schall, H.-L. Truong, and S. Dustdar. Unifying Human and Software Services in Web-Scale Collaborations. *Internet Computing*, 12(3):62–68, May-June 2008.

[14] F. Skopik, D. Schall, and S. Dustdar. Modeling and mining of dynamic trust in complex service-oriented systems. *Information Systems*, 35:735–757, 2010.

[15] R. Sterritt. Autonomic computing. *ISSE*, 1(1):79–88, 2005.

[16] M. Treiber, L. Juszczyk, D. Schall, and S. Dustdar. Programming Evolvable Web Services. In *ICSE*, pages 43–49, 2010.