Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

Tool Suite for Deployment, Monitoring & Controlling of Virtual Service Platforms (Interim)

Abstract

In order to bring Virtual Service Platforms to the operational state, besides its pre-configuration, additional deployment and runtime activities need to take place.

To ensure the proper execution of the VSP we advocate the use of several supporting tools for deployment, monitoring and controlling whose design and implementation is challenging for several reasons: (i) monitoring and adaptation tools have to take into account heterogeneousness of underlying service platforms; (ii) different service platforms use different mechanisms and communication schemes for monitoring (i.e. events, JMX, SMNP) from which all needs to be covered by appropriate tools; (iii) it is not trivial to define a standardized monitoring and adaptation interfaces for all existing and future service platforms.

This deliverable shows the interim state of the tools proposed to address these challenges and draws the next steps which will be fulfilled during next periods as the project evolves.

Document ID:	INDENICA – D4.2.1
Deliverable Number:	D4.2.1
Work Package:	WP4
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2012-04-23
Author(s):	TEL

Project Start Date: October 1st 2010, Duration: 36months

Version

0.1	1. February 2012	initial version
0.2	23 February 2012	Structure and initial input
0.3	19 March 2012	Additional tools' description
0.4	29 March 2012	Final version for the internal review
0.5	2 April 2012	Approve and release
1.0	23 April 2012	Final Version

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

1 Introduction 6
2 Integration of the tool suite with supporting tools 7
2.1 Goal-based modelling Framework 7
2.2 Service Platform Infrastructure Repository and EASy-Producer 7
2.3 Monitoring rule editor 8
2.4 Adaptation rule editor 8
2.5 View-based Modelling Framework 9
2.5.1 Generation of monitoring rules based on SLA modelling 9
2.5.2 Generation of deployment descriptors 10
3 Tool suite for deployment of Virtual Service Platforms 11
3.1 Deployment Manager 11
3.2 Example of the Deployment Process 11
4 Tool suite for monitoring of Virtual Service Platforms 14
4.1 SPASS-meter 14
4.2 ECoWare 17
4.3 Monitoring engines 19
4.4 Domain-specific events 20
5 Tool suite for adaptation of Virtual Service Platforms 24
5.1 Adaptation Engines 24
5.2 Improvement of Monitoring and Adaptation 24
5.3 Adaptation of base platforms 27
6 Summary and future work 29
A Appendix 1: EcoWare Usage Guide 30
A.1 Installation and Setup 30
A.2 Example 32
B Appendix 2: SPASS-meter Quick Guide 35
B.1 Installation 35
B.2 Setup 35
B.3 Example 36
C Appendix 3: Indenica Runtime Platform Demonstrator 38
C.1 Initial Eclipse Project Setup 38
C.2 Getting Project Dependencies 38

C.2.1 Manual Installation of Dependencies 38
C.2.2 Use a preconfigured Virtual Machine with all Dependencies 39
C.3 Starting the Platform.. 39
C.3.1 Command Line 39
C.3.2 Eclipse... 40
References 41

Table of Figures

Figure 1: Monitoring Rule Editor showing exemplary monitoring rule 8
Figure 2: Adaptation Rule Editor showing exemplary adaptation rule........................ 9
Figure 3: Architecture of SPASS-meter... 15
Figure 4: Architecture of the ECoWare Framework.. 18
Figure 5: Runtime Infrastructure Architecture: Monitoring Overview 19
Figure 6: Runtime Infrastructure Architecture: Adaptation Overview 24
Figure 7: The SOA Governance Vitality Method 25

1 Introduction

The main purpose of the tool suite proposed as a part of this Work Package is to bring the models, mechanisms and tools delivered by Work Packages 1-3 into their operational state producing Virtual Service Platforms that will be deployable and manageable at runtime.

Different tools make it possible to create scalable monitoring mechanisms that allow generating a global overview of the VSP.

Tool suite also allows actors to cope with non-functional metrics of underlying service platforms for different reasons, for example: (i) drilling down for root cause of performance bottlenecks; (ii) estimation and evaluation of current and planned adaptation policies; (iii) evaluation and reacting based on global view of the runtime environment.

In the second section of the document we describe how we leverage tools provided by different technical Work Packages to support the WP4 tool suite. In the third section we describe the initial proof-of-concept deployment tool for instantiating different heterogeneous service platforms together with their added-value interfaces. Fourth section describes three tools that provide monitoring capabilities. First two of them concentrate on non-functional parameters and are complementary to each other providing general views of the VSP from platform and service points of view. In the same section we also describe how the monitoring engines cooperate with different service platform and what kind of mechanisms they do use. In the fifth section we concentrate on the tools that support adaptation of service platforms both on service level and on platform level. Adaptation described in this section also entails certain changes in the monitoring engine execution; therefore we describe the dynamism of this approach in Section 245.2. In the last section we provide a summary of the current stage of the tool suite for Deployment, Monitoring & Controlling of Virtual Service Platforms and give a view on the next steps regarding further development.

2 Integration of the tool suite with supporting tools

2.1 Goal-based modelling Framework

The elicitation of the requirements of the virtual platforms will be carried out through IRET (INDENICA Requirements Elicitation Tool). IRET is the Eclipse-based framework that supports IRENE (INDENICA Requirements ElicitatioN mEthod). IRENE is a goal-based requirements elicitation approach that blends goals, adaptation capabilities, and variability into a single coherent solution.

Functional and non-functional requirements are rendered through goals. Adaptation capabilities, at requirements level, are specified through special-purpose goals called *adaptation* goals, and variability is added in the form of textual comments entered through particular forms. Different goal models (coming from different applications or from the viewpoint of different stakeholders), or different views on the same model can then be merged by means of simple syntactic tools that help the user merge commonalities, highlight differences, and negotiate among the different alternatives. Interested reader can refer to deliverable [D1.2.1] for a complete presentation of the approach.

As for the tool, IRET is implemented on top of GMF/EMF and provides the user with the usual modelling capabilities required to elicit and specify requirements. Users can adopt an informal approach towards requirements, and thus associate simple textual comments to the different elements, or a more detailed and formal view on requirements, and thus also exploit the formal languages supplied by IRENE.

Produced models, that is, requirements specifications, can be used and handled in two different ways. Since IRET is an Eclipse plug-in, based on EMF, each concept is materialized in a specific object within the system and can be accessed through the standard interfaces provided by Eclipse. In addition, models can also be serialized into XML documents, and then be manipulated through the “well-known” tools. These two options are also the two ways IRET, and its artefacts, can be integrated into the complete INDENICA tool-suite.

2.2 Service Platform Infrastructure Repository and EASy-Producer

To support the creation and maintenance of VSPs, the Service Platform Infrastructure Repository encapsulates design time, deployment time, as well as runtime aspects of service platforms. When the deployment of a concrete service platform instance happens, all variabilities with a binding time of at latest the deployment time have been resolved. The variability resolution is done using the EASy-Producer tools developed in WP2 in an interactive fashion either during development, i.e. before deployment, or, for deployment time variabilities during the deployment process. Thus, at deployment time a partially instantiated variability model which only contains open variabilities to be resolved at runtime is available. This partially instantiated model is used to populate the Service Platform Infrastructure Repository of the service platform instance in order to guide the monitoring and adaptation engines. The repository runtime component is deployed with a service platform instance and contains information about the models used to

generate and create the VSP instance, in addition to variability and deployment configurations. Furthermore, monitoring and adaptation rules, created using the accompanying 'Adaptation rule editor' and 'Monitoring rule editor' tools, are stored in the repository for later use by the monitoring and adaptation engines respectively. Moreover, monitoring data generated by the VSP instance is stored the runtime repository for later analysis. The Service Platform Infrastructure Repository is described in greater detail in Deliverable [D2.3.1].

2.3 Monitoring rule editor

The Monitoring Rule Editor is used to manage monitoring rules stored in the runtime repository component.

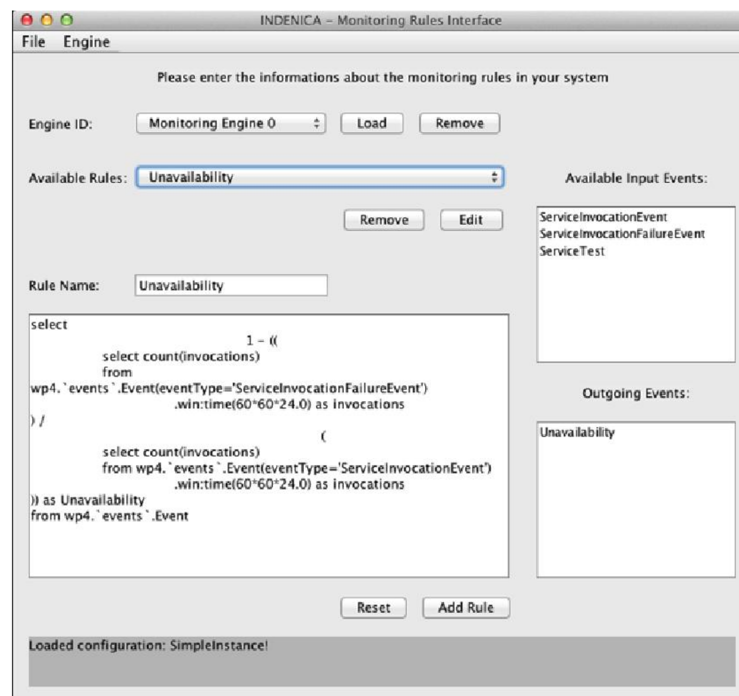


Figure 1: Monitoring Rule Editor showing exemplary monitoring rule

Monitoring rules can either be generated from the VbMF or created manually, and are used by the Monitoring Engines to efficiently and effectively gather runtime data from the integrated service platforms as well as the VSP instance. The Monitoring Rule Editor is described in greater detail in Deliverable [D2.3.1].

2.4 Adaptation rule editor

The Adaptation Rule Editor is used to manage adaptation rules stored in the runtime repository component.

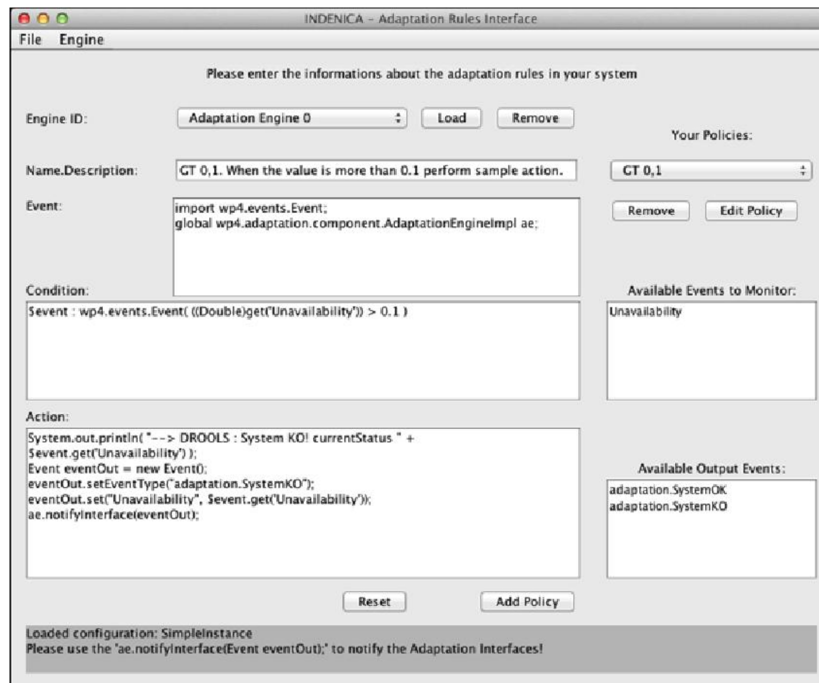


Figure 2: Adaptation Rule Editor showing exemplary adaptation rule

Similar to monitoring rules, adaptation rules can either be generated from the VbMF or created manually, and are used by the Adaptation Engines to effectively adapt the VSP instance and integrated service platforms to changes in the environment in order to always maintain the best possible performance, availability and/or scalability according to business requirements. The Adaptation Rule Editor is described in greater detail in Deliverable [D2.3.1].

2.5 View-based Modelling Framework

The integration of View-based Modelling Framework (VbMF) [D3.1] with the tool suite presented in this document is to leverage VbMF views in order to (semi-)-automatically generate rules for the monitoring component mentioned in Section 4 and deployment descriptions for the deployment component mentioned in Section 3.

2.5.1 Generation of monitoring rules based on SLA modelling

VSPs and service platforms will be monitored at runtime in order to track the health of the system or to measure certain metrics. The view-based code generation tools described in [D3.1] and the Runtime View are used to generate rules used by the Monitoring Engine. In Section 5.5 of [D3.1], we presented in detail how VbMF views and techniques can support to describe and generate event-based rules that can be fed to some complex event processing framework such as Esper[1] for monitoring systems' and services' SLA properties such as availability, execution time, response time, to name but a few. The main goal of the integration of VbMF and the monitoring component is to implement a richer low-level view that refines the

[1] http://esper.codehaus.org

concepts of the aforementioned Runtime View and, based on this low-level view, generates rules and directives in Esper for monitoring the events occurring in VSPs and/or, partially, generates necessary code for monitoring VSP services.

2.5.2 Generation of deployment descriptors

VbMF Deployment View provides high-level concepts such as the artefacts to be deployed and the nodes where the artefacts are hosted. The aforementioned Deployment View will be refined down to support relevant deployment concepts for the corresponding runtime environment where INDENICA VSPs are executed, which is Apache Tuscany 1.x. In particular, artefacts to be deployed are SCA composites each of which can comprise a number of service components (i.e., SCA components). The SCA 1.0 specification [OSOA] allows an SCA Composite and related artefacts, for instance, its implementation and its required interfaces and components, to be grouped and deployed in a managed unit called *contributions*. A logical group of such SCA contributions that forms an area of business functionality controlled by a single organisation, for instance, whole of a business or a department within a business [OSOA] can be deployed and managed in a larger unit called SCA *domain*. The main goal of the integration of VbMF and the deployment component mentioned in Section 3 is to implement a low-level deployment view that refines the aforementioned high-level concepts and targets the deployment of SCA, and especially Apache Tuscany. Based on the low-level deployment view, we can generate SCA composite and contribution descriptions that assemble corresponding components of the VSP under consideration and successfully deploy those for executing in an SCA runtime (i.e., Apache Tuscany 1.x)

3 Tool suite for deployment of Virtual Service Platforms

3.1 Deployment Manager

The virtual service platform has to be deployed in a runtime environment. The deployment consists thereby of three major steps as described in Deliverable D4.1. First, all artifacts have to be packaged into a deployable format. Second, these artifacts have to be uploaded to a (remote) location, so they are available for use in the runtime. The third and last step of deployment is the registration of the artifacts in the runtime, so they are known, can be wired together and are available for other services. For the deployment manager to work, several prerequisites have to be fulfilled:

Maven. As build management tool, Maven is used. A typical VIP-Project will consist of a multi-module maven project. Every module in this project describes a contribution in the sense of SCA, consisting of artifacts, composite descriptions and contribution metadata.

Tuscany. For INDENICA, the Tuscany SCA Java runtime will be used, implementing the OSOA SCA specification 1.0. During the deployment process, information about a running Tuscany instance has to be supplied, e.g. server address, user credentials etc.

SCA Deployment Configuration. Information about every SCA composite has to be supplied. Tuscany needs every composite to be run on a single Node (a Tuscany runtime itself). These composites have to be registered at some Endpoint, so its services are made available. This information is supplied as parameters in the maven descriptor (pom.xml).

The deployment process is initiated with the command `mvn indenica:deploy`, run at the parent project after the packaging phase. Every child project then will be deployed as follows: The artifacts are uploaded to the Tuscany server and are started in the same runtime. After uploading, the artifacts are scanned for contribution metadata and all deployable composites are registered in the Tuscany domain. For every composite, a node will also be registered and started. The address of the node will thereby be determined by the SCA deployment configuration.

3.2 Example of the Deployment Process

The following example describes a sample warehouse application that offers several services, implemented in java. The whole setup consists of two projects, the maven parent project and a maven project for compiling and packaging the services. Details about the Tuscany platform to which to deploy this application are provided in the pom.xml of the parent project:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sca.indenica</groupId>
  <artifactId>parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
```

```
  <name>parent</name>

  <modules>
    <module>../warehouse</module>
  </modules>

  <build>
  <plugins>
    <plugin>
     <groupId>com.sap.research</groupId>
     <artifactId>indenica-maven-plugin</artifactId>
     <version>1.0</version>
     <configuration>
      <tuscany>
        <server>127.0.0.1</server>
        <mgmtPort>9990</mgmtPort>
      </tuscany>
     </configuration>
    </plugin>
  </plugins>
  </build>

</project>
```

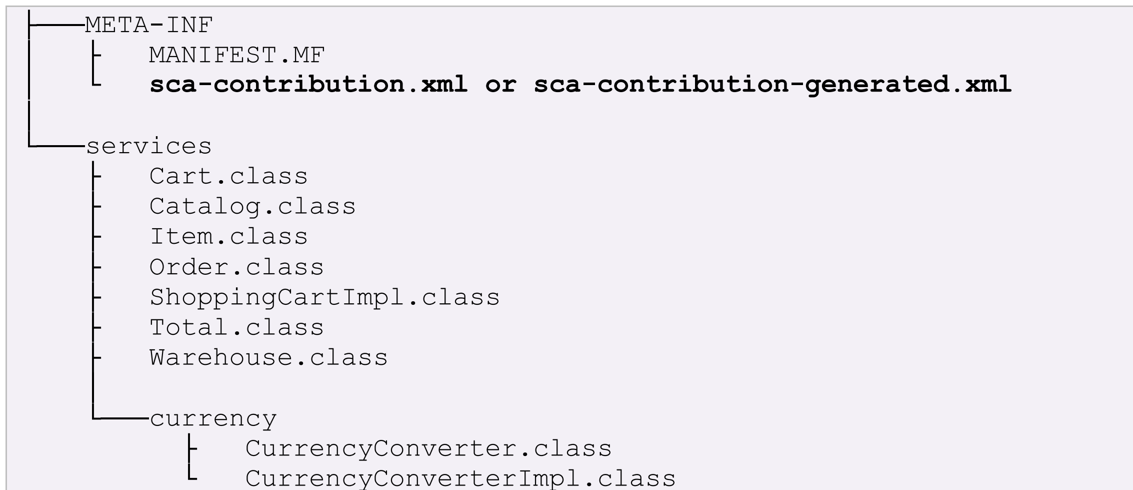
The domain server as well as the port of the management interface has to be described in the configuration part of the indenica-maven-plugin (in the parent project). This implicitly configures the Tuscany target for every child module of this maven build.

As described earlier, every composite will run in a separate node. The configuration of this node will be expressed in the pom.xml of every child module. The following section of the pom.xml of the *warehouse* module configures the endpoints for the services:

```
<plugin>
  <groupId>com.sap.research</groupId>
  <artifactId>indenica-maven-plugin</artifactId>
  <version>1.0</version>
  <configuration>
   <scaNode>
     <ports>
       <catalogs>8101</catalogs>
       <currency>8102</currency>
     </ports>
   </scaNode>
  </configuration>
 </plugin>
```

The described services must correspond to the defined names for each composite. This configuration is only necessary, if composites exist. The SCA composite descriptions have to reside in the root of the archive to be deployed. A typical packaged Indenica SCA project structure looks like the following example:

```
warehouse.jar
├   catalogs.composite
├   currency.composite
├   warehouse.html
│
```



```
├───META-INF
│   ├   MANIFEST.MF
│   └   sca-contribution.xml or sca-contribution-generated.xml
│
└───services
    ├   Cart.class
    ├   Catalog.class
    ├   Item.class
    ├   Order.class
    ├   ShoppingCartImpl.class
    ├   Total.class
    ├   Warehouse.class
    │
    └───currency
          ├   CurrencyConverter.class
          └   CurrencyConverterImpl.class
```

Important for the deployment process are the bold files. The deployment manager will process each of these files to gain enough information for the deployment activity.

Running the `mvn indenica:deploy` command results in the deployment of the two composites *catalogs* and *currency* to the nodes with the endpoints 127.0.0.1:8101 rsp. 127.0.0.1:8102.

To undeploy the composites, run the `mvn indenica:undeploy` command on the parent project.

4 Tool suite for monitoring of Virtual Service Platforms

4.1 SPASS-meter

SPASS-meter[2] is a flexible resource monitoring framework which enables observing the resource consumption of individual parts of a software system at runtime. In INDENICA, SPASS-meter will monitor the resource consumption of individual services, in particular those which are configured and adapted at runtime. SPASS-meter complements the INDENICA runtime environment and enables the adaptation manager to choose among alternative available services based on their actual resource consumption. In contrast, typical resource monitoring systems provide either large amounts of detailed information which is not related to individual services (e.g. InsECT [CO04], J-RAF2 [BH04] or OpenCore [JI12]) or which is aggregated on system level (e.g. JMX [O08]) representing summarized effects of all running services. In this section, we will give an overview of SPASS-meter, its workflow, its configuration, the supported resources, its architecture and its planned integration into the INDENICA runtime environment.

Monitoring resource consumption with SPASS-meter happens according to the following workflow: SPASS-meter needs to be configured for a specific system under monitoring (SUM, in INDENICA a service platform instance) in order to gather information on the services of interest. Based on that configuration, the SUM is then prepared for monitoring prior to or at runtime. At runtime of the SUM the SPASS-meter framework collects information on basic resource consumptions, e.g. the allocation of a portion of memory, and aggregates the information as previously configured. The obtained information may be summarized at the end of the SUM (post mortem analysis) or passed to subsequent processing, e.g. to the INDENICA monitoring and adaptation engines.

Observing low-level information from a SUM typically implies a certain overhead as additional functionality must be executed to obtain, collect, aggregate and even analyze the low-level information. In order to focus on relevant information and, as a side effect, to reduce the monitoring overhead, SPASS-meter allows specifying the **monitoring scope**. The monitoring scope specification defines what is measured at which level of detail, i.e. which resources are observed for which services. A service is specified based on its service interface, i.e. the classes and methods used from outside to access the service. SPASS-meter further allows specifying whether dependent functionality of a service should be accounted for that service. For a specific SUM, the monitoring scope can either be specified as source code annotations or as an XML configuration file.

[2] SPASS is the acronym for Simplifying the Development of Adaptive Software Systems and SPASS-meter is one of the foundational building bricks of our approach in this field. In German, the term “Spass” means “fun” and points to the tons of fun the developers had and will have while realizing the approach and its tooling.

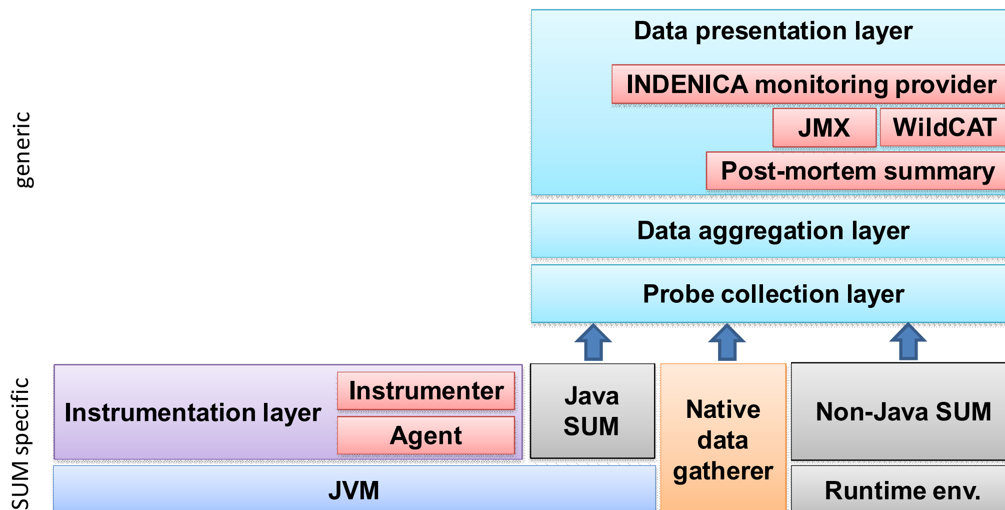


Figure 3: Architecture of SPASS-meter

SPASS-meter supports the following **resource types**:

- Execution time, i.e. the amount of time consumed by the CPU for executing a service. For an entire service, the execution time is measured as (threaded) CPU time. In contrast, response time measures the system time consumed by executing one (or more) individual operations provided by a service.
- Memory consumption in terms of memory allocation or memory usage. Memory allocation refers to the total amount of memory requested by a service. In contrast, the memory usage of a service is the amount of allocated but yet not freed memory.
- File transfer - the number of bytes read from or written to files including administrative overhead.
- Network transfer is the number of bytes read from or written to a network interface including administrative overhead.

Dependent on the monitoring scope specification, SPASS-meter is also capable of obtaining and monitoring the related resource consumption of the entire SUM or the underlying operating system process, e.g. to break down the process load to individual services.

The monitoring scope specification, i.e. the configuration of relevant services and resources to be monitored is an important mechanism to achieve flexibility and to balance monitoring overhead. We will discuss further mechanisms such as preparing the SUM for monitoring prior to or at runtime along with the **architecture** of SPASS-meter. As depicted in Figure 3, SPASS-meter consists of a SUM-specific part (which depends on the runtime environment and the type of SUM) and a generic part which can be reused for different types of SUM. We will start off with the SUM-specific part.

- Runtime environment: The runtime environment provides mechanisms to execute the SUM. For a Java SUM, the runtime environment is a virtual machine (VM), e.g. the Java Virtual Machine (JVM) running a service platform. For a native SUM the runtime environment is the operating system.

In order to access information from the operating system such as process load in a uniform way, SPASS-meter uses a cross-platform low-overhead native library developed by SUH (named "Native data gatherer" in Figure 3). The embedding of that native library itself can be configured so that existing data from the runtime environment, e.g. via JMX can directly be used. In this deliverable we focus on Java SUMs (support for non-Java SUMs is currently being developed).

- Instrumentation layer: This layer prepares the SUM for monitoring, i.e. for inserting calls into the SUM which notify the SPASS-meter framework about basic resource consumptions. The insertion of the notification calls may happen dynamically during runtime, statically before runtime or in mixed fashion. In the static case, the instrumentation layer is triggered as an external tool for a given set of classes. In the dynamic case, the JVM informs the so called instrumentation agent about classes being loaded and, in turn, the agent triggers the instrumentation. In SPASS-meter, instrumentation actions are realized in the instrumentation layer as abstract strategies while concrete instrumentation is performed by a pluggable instrumentation library (named "Instrumenter" in Figure 3).

The result of applying the SUM specific part of SPASS-meter is an instrumented SUM which is prepared to inform the generic part of the SPASS-meter framework about resource consumption. Now we discuss the layers of the generic part:

- Probe collection layer: This layer receives the notifications about basic resource consumptions from the instrumented SUM and synchronizes the processing of subsequent layers. This layer provides a platform-independent interface so that non-Java SUMs can be supported in a uniform way.
- Data aggregation layer: In this layer the individual resource consumptions are aggregated as defined in the monitoring scope specification, i.e. the actual resource consumption for individual services is determined. This layer also provides support for dynamic software product lines, e.g. resource consumption is assigned to individual configurations of enabled variabilities.
- Data presentation layer: Finally, the collected data is presented for external further processing. SPASS-meter provides interfaces for obtaining periodic runtime events or post-mortem summaries. Several plugins implementing the interfaces are already realized such as a JMX integration, an integration with WildCAT [OW2C10] or a textual post mortem summary.

The integration into the INDENICA runtime platform exploits the flexibility and openness of SPASS-meter. As stated above, SPASS-meter will provide runtime resource consumption information on the variabilities of service platform instances, particularly for alternative variabilities.

- The monitoring scope specification is generated as an XML file from the partially instantiated variability model, i.e. from the runtime variabilities. As described in Deliverable D4.1, the scope configuration is derived as one preparation step of the deployment process based on the instantiated

variability model (WP 2) which is stored in the Service Platform Infrastructure Repository (see also Section 2.2).

- The service implementation is prepared for monitoring based on the generated monitoring scope specification. This happens as part of the deployment process described in Deliverable D4.1. SPASS-meter is applied in static instrumentation mode and deployed with the service platform instance and may run in mixed instrumentation mode if required, e.g. due dynamically loaded or structurally reconfigured services.
- SPASS-meter is integrated as part of the monitoring interface in the service platform instance. Therefore, as a specific plug-in on the SPASS-meter data presentation layer produces periodic INDENICA specific monitoring events realized. An alternative to be explored is the tight integration into the EcoWare/Siena P/S bus (see Section 4.2) which may allow reducing the communication load between service platform instance and the INDENICA runtime environment.

The core concepts of SPASS-meter have been designed and realized prior to INDENICA. Within INDENICA, SPASS-meter was extended, adapted and optimized for use and integration into the INDENICA runtime environment. Currently, SPASS-meter is available for Java SUMs and equipped with an initial event-based INDENICA monitoring interface integration. Monitoring of non-Java SUMs is under development.

In this section we described the SPASS-meter monitoring approach, its configuration using a monitoring scope specification, the supported resources, its capabilities, its architecture as well as its integration into the INDENICA runtime environment. In particular, a focused monitoring scope definition obtained from the INDENICA variability model and running SPASS-meter in static (or even mixed) mode will balance deep insights into service resource consumptions, flexibility and monitoring overhead.

The documentation about usage, installation and setup of the SPASS-meter is available as Appendix 2: SPASS-meter Quick Guide to this document.

4.2 ECoWare

ECoWare, which stands for Event Correlation Middleware, is a distributed data aggregation and persistency tool. In INDENICA ECoWare is used to provide means to reason on a complex service-based system, by aggregating raw data collected from its multiple layers. It allows us to correlate behaviours being seen in the Service Platform Environment with behaviours being seen at the underlying virtual resources layer.

A typical ECoWare deployment (see Figure 4) consists of four different types of components: (i) the execution environments for which we want to collect run-time data (together with appropriate probes), (ii) a series of processors for providing the actual data aggregations, (iii) a persistency database, and (iv) the ECoWare Dashboard for visualizing the aggregated data. The components collaborate through a Siena Publish and Subscribe (P/S) event bus for which ECoWare defines a

normalized event format. In order to manage the normalized format and to collaborate properly, each component is required to implement appropriate SienaInputAdapters and SienaOutputAdapters.

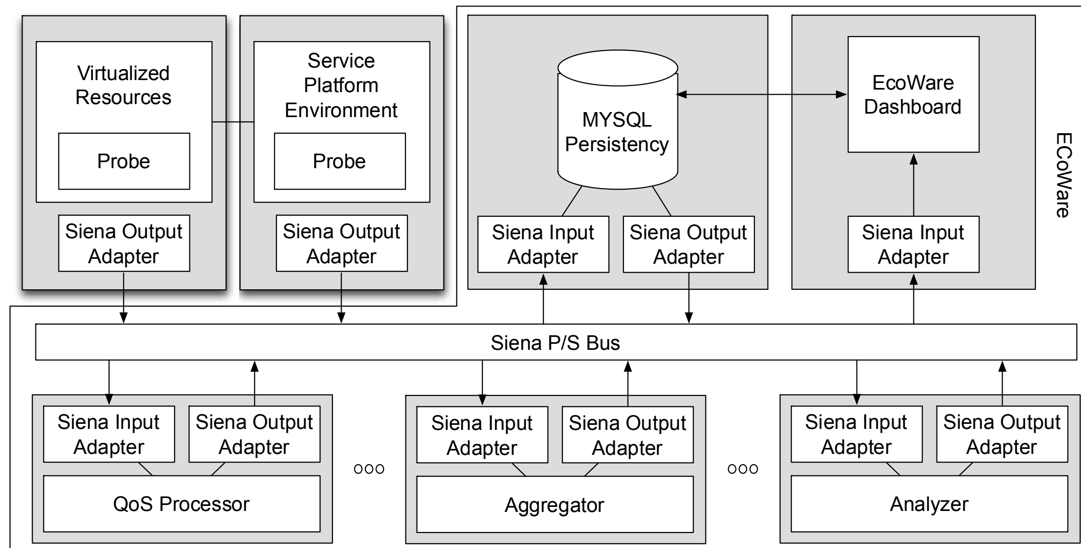


Figure 4: Architecture of the ECoWare Framework.

The execution environments can be of any kind. In Figure 4 we are probing the Service Platform Environment, as well as the underlying virtualized resources on which it is run. This allows us to aggregate information coming from both, with the goal of incrementally constructing comprehensive knowledge of the running system along its multiple layers.

Data processing is provided by three different kinds of processors QoS Processors, Aggregators, and Analyzers. These processors subscribe to events on the P/S bus, execute their internal logic, and publish their results back to the bus. This allows them to be strung together to obtain the aggregated information we desire. Thanks to this loose coupling ECoWare can be considered very extensible. In the future we will investigate a stronger integration with the other monitoring tools produced in the context of the Indenica project.

QoS Processors and Aggregators are built using Esper, a component for complex event processing activities. Esper components execute queries defined using an Event Processing Language (EPL). EPL is an SQL-like language for developing event conditions, correlations, and aggregations. The difference between SQL and EPL is that, instead of running queries against stored data, Esper stores queries and runs data through them. The execution model is thus continuous rather than limited to the exact moment at which the query is submitted. Analyzers are built using a simple assertion analyzer inspired by our previous work on WSCoL [WSCoL]. WSCoL was developed for the definition of functional monitoring of BPEL processes, and has been modified to receive data that are not in XML form.

The ECoWare dashboard is a Java Desktop application that system managers can use to visualize data collected through ECoWare. It supports both live charting, and online and offline violation drill-down analysis. Online and offline drill-down analysis allows managers to choose a violation, and visualize the multiple data collections that were triggered by that event. Violations, as well as the correlated data are collected on-demand from ECoWare's persistency database, where they are automatically collected every time an aggregator is triggered. Like for certain security cameras, we currently store data for 24 hours, and periodically cleanse the data that are no longer needed.

4.3 Monitoring engines

The INDENICA infrastructure provides an extensible platform monitoring framework employing novel concepts for organizing and layering monitoring concerns to allow for efficient distribution of software components to reduce communication overhead. Furthermore, sophisticated processing methods, such as data ageing, allow for the effective usage of historical system health data while keeping transmission and storage overhead minimal.

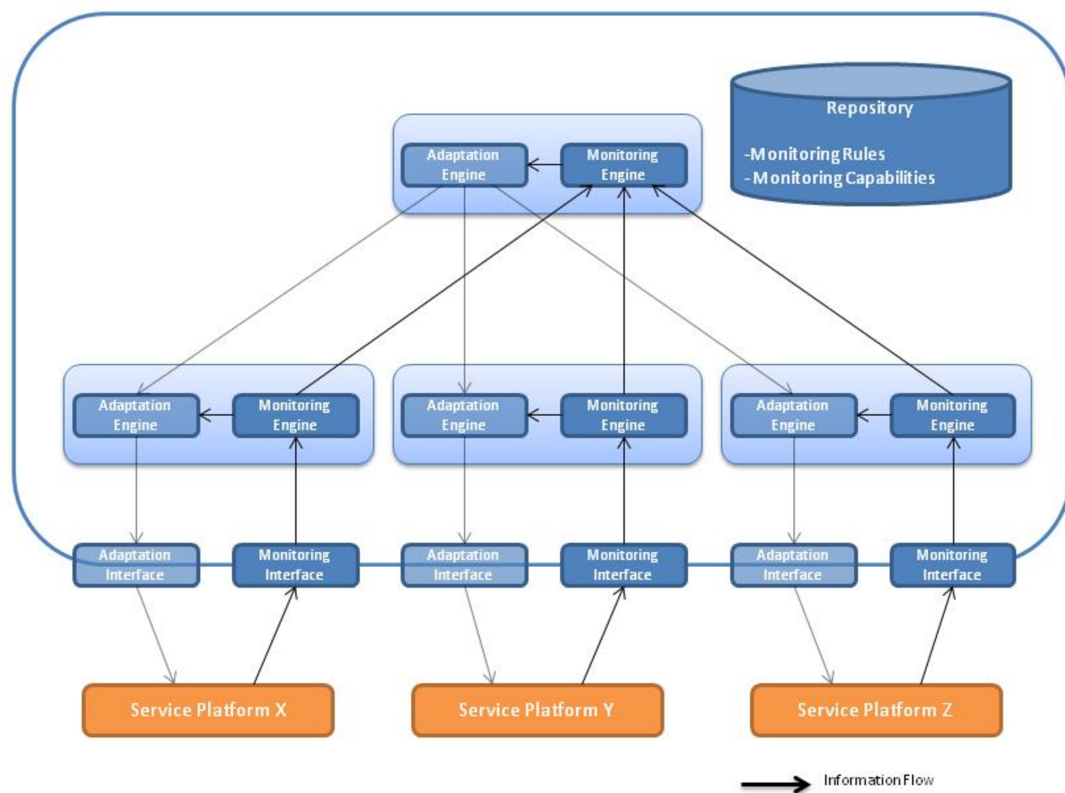


Figure 5: Runtime Infrastructure Architecture: Monitoring Overview

The monitoring infrastructure is instantiated according to configuration directives deployed to the repository (cf. Section 2.2) by creating all necessary monitoring engines and establishing connections to the integrated service platforms. Additionally, connections between monitoring engines and according adaptation engines (cf. Section 5.2) are set up.

4.4 Domain-specific events

Events monitored in the VSP come from different service platforms. In order to allow efficient monitoring of events the necessary unification of these events has to be done. INDENICA proposes an extensible event model to which all underlying platforms need to be compliant with to catch all messages. The team identified three major points of unification.

First is the format that is used to define messages, the second is the internal structure of these messages and third is the type of messaging technique that is used to gather events. Events that are sent by the underlying service platforms may be written in different formats such as JSON [JSON], XML [XML], key-value list, etc. and several mechanisms can be used to get them, f.e. publish-subscribe, polling, streaming, etc.

The first Use Case example that we use is the Remote Maintenance System which uses XML-based events which are directly sent to the monitoring engine using publish-subscribe queuing mechanism. These events are compliant to the following XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Event" type="EventType" />
  <xs:complexType name="EventType">
    <xs:sequence>
      <xs:element name="EventName" type="xs:string" />
      <xs:element name="callerSIPId" type="xs:string" minOccurs="0"/>
      <xs:element name="callParameters" type="xs:string"
minOccurs="0"/>
      <xs:element name="sessionId" type="xs:string" minOccurs="0"/>
      <xs:element name="responseMsgId" type="xs:int" minOccurs="0"/>
      <xs:element name="errorMsgId" type="xs:int" minOccurs="0"/>
      <xs:element name="message" type="xs:string" minOccurs="0"/>
      <xs:element name="userName" type="xs:string" minOccurs="0"/>
      <xs:element name="userPasswd" type="xs:string" minOccurs="0"/>
      <xs:element name="userGroup" type="xs:string" minOccurs="0"/>
      <xs:element name="statusId" type="xs:int" minOccurs="0"/>
      <xs:element name="userParameters" type="xs:string"
minOccurs="0"/>
      <xs:element name="newUserId" type="xs:string" minOccurs="0"/>
      <xs:element name="userSipId" type="xs:string" minOccurs="0"/>
      <xs:element name="Response" type="ResponseMsg" minOccurs="0"/>
      <xs:element name="User" type="UserData" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="ResponseMsg">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="responseMsgId" type="xs:int" minOccurs="0"/>
      <xs:element name="errorMsgId" type="xs:int" minOccurs="0"/>
      <xs:element name="message" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="UserData">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="userId" type="xs:string" minOccurs="0"/>
      <xs:element name="userName" type="xs:string" minOccurs="0"/>
      <xs:element name="userPasswd" type="xs:string" minOccurs="0"/>
      <xs:element name="userGroup" type="xs:string" minOccurs="0"/>
      <xs:element name="statusId" type="xs:int" minOccurs="0"/>
      <xs:element name="userParameters" type="xs:string"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

This XML schema corresponds to events generated by Remote Maintenance subsystem during its functional operations. It consists of main Event element with several optional parameters and two complex elements with information about corresponding users (i.e. call recipients) and responses corresponding to complex request (i.e. adding several users to call session).

The second subsystem that was used to derive a common event model was Yard Management Subsystem, which is sending events formatted based on JSON. Gathering events from this subsystem requires polling to the REST-based endpoint, which in result sends events that were not previously fetched.

The Format of events is a simple 4 attribute JSON structure:

```
{
"id" :"<id>",
"triggered":<timestamp>,
"type":"<string>",
"data":{<object>}
}
```

The description of types and inner structure of the data element can be found in the Table 1 and Table 2.

The target unified event model used by INDENICA will be derived from base service platforms. After that each of base platforms will need to be tailored to comply with the model and the communication mechanisms used in the monitoring engine.

These unified models will base a monitoring interface, which will be developed in the second half of the project.

	type	description	data
Trailer	TruckCheckIn	checkin of truck at checkpoint	appData
	TruckCheckOut	checkout of truck at checkpoint	appData
	TruckDelayed	A truck reports a delay	appData
	TruckRescheduling	tresheduling due to delays of trucks or delays at dock	appData
Jockey	TaskScheduled	a task was created for a jockey	taskData
	TaskReserved	a jockey reserved a task	taskData
	TaskFinished	a task was finished by a jockey	taskData
	TrailerRelocation	additional event on finished trailer reloc.	taskData
Errors	MisdirectedTrailer	if a trailer finds itself at a wrong dock	trailerId, appData
	scheduling collision	collision in schedule due to delays	appData1, appData2
	NoAppointmentFound	if no appointment could be found for a req. Appointmentdate	appData
	NoFreeWaitingbay	if no waiting bay could be found for a check-in-truck	appData

Table 1: Internal structure of JSON-based event of Yard Management Subsystem

appData	appointmentId, start, end, dock, truckId
taskData	timestamp, taskType, trailerId, jockeyId, origin, destination

Table 2: Inner structure of data elements of Yard Management Subsystem event model

The third system was the warehouse system which contains two subsystems, the warehouse management system and the conveyor control subsystem. Both subsystems add events to message queues.

The **warehouse management system** supports events related to transportation units and events related to orders. Events are sent as text based key value pairs.

Order related events

```
MessageType : OrderCreated | PickingStarted | OrderFinished
OrderId : <alphanumerical identifier>
```

Transport unit related events

```
MessageType : TransportUnitCreated | TransportUnitStored
TransportUnitId : <alphanumerical identifier>
```

The Conveyor control system supports events related to conveyors and events related to stacker cranes:

Conveyor related events:

```
MessageType : ConveyorTransportStarted
```

```
TransportUnitId : <alphanumerical identifier>
```

```
MessageType : ConveyorTransportFinished
TransportUnitId : <alphanumerical identifier>
TransportUnitLocation : <alphanumerical identifier>
```

Stacker crane related events

```
MessageType : StackerCraneStartedPickingUp
CraneId : <alphanumerical identifier>
TransportUnitId : <alphanumerical identifier>
```

```
MessageType : StackerCranePickedUp
CraneId : <alphanumerical identifier>
TransportUnitId : <alphanumerical identifier>
```

```
MessageType : StackerCranePickedUp
CraneId : <alphanumerical identifier>
TransportUnitId : <alphanumerical identifier>
TransportUnitLocation : <alphanumerical identifier>
```

5 Tool suite for adaptation of Virtual Service Platforms

5.1 Adaptation Engines

In concert with the monitoring infrastructure, INDENICA provides an extensible, layered platform adaptation framework, geared towards efficient and effective control of service platforms, minimizing communication overhead while maintaining high flexibility and allowing for complex management structures.

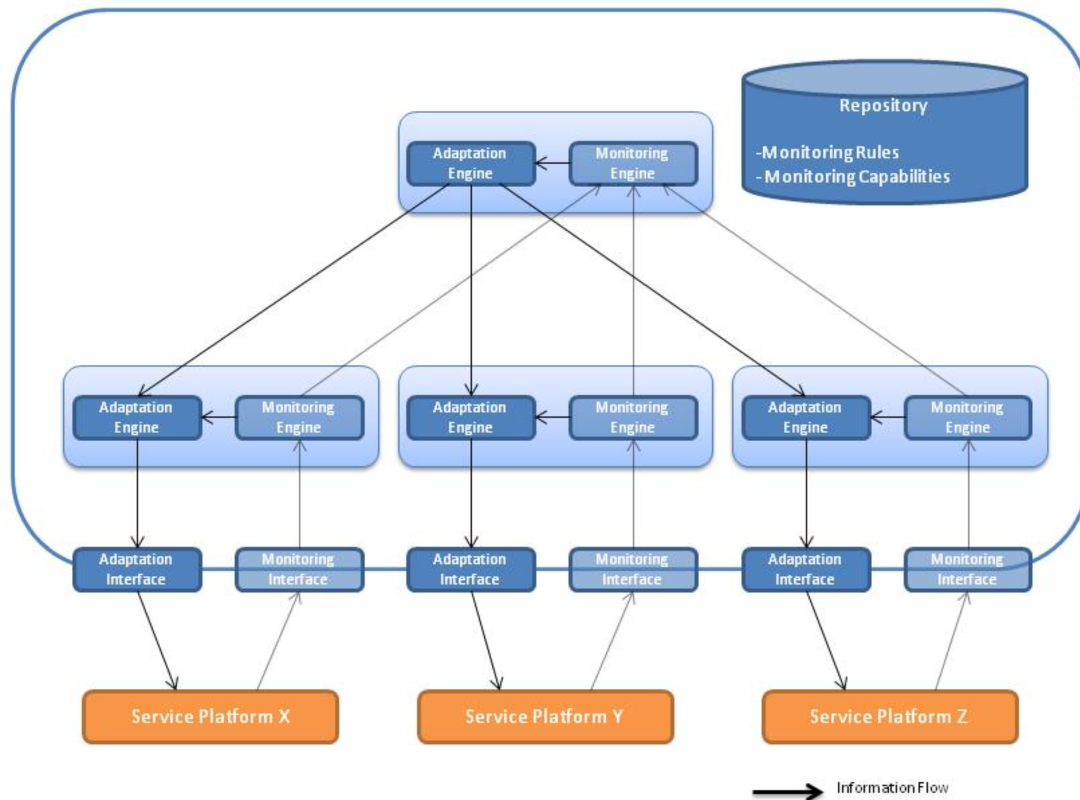


Figure 6: Runtime Infrastructure Architecture: Adaptation Overview

The adaptation infrastructure is instantiated according to configuration directives deployed to the repository (cf. Section 2.2) by creating all necessary adaptation engines and establishing connections to the integrated service platforms. As mentioned in Section 4.3, connections between adaptation engines and according monitoring engines are set up.

5.2 Improvement of Monitoring and Adaptation

While adaptation in the above described chapter is an automated process that works within limits given by the platform architecture, functionality and variability, there will always be monitoring results that trigger actions in the direction of platform evolution and improvement of the measurement definitions behind the monitoring and adaptation rules.

In the first case the analysis of monitoring results identifies the need of changing the platform, implementing and deployment of a new release. For this purpose, the INDENICA Governance (see [D3.2]) defines two processes:

- A change and error management process for controlling changes at the platform during the development and operation phase
- A platform portfolio process for controlling strategic planning and coordinated development of platforms, services and applications.

Here we want to have a closer look on the improvement of measurements, monitoring rules and adaptation rules.

The improvement cycle of the SOA Governance Framework of The Open Group is called SAO Governance Vitality Method (see [SGF 2009] page 32). It aims at improving the overall governance system containing policies, processes, guidelines and standards, roles and responsibility allocation.

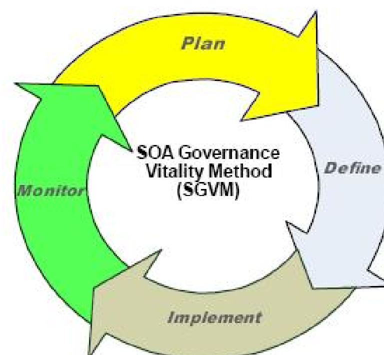


Figure 7: The SOA Governance Vitality Method

This model with its four phases can also be applied to the continuous improvement of measurements, monitoring and adaptation.

The triggering situations for such improvement can be:

- A monitoring rule does not create events at all.
- A monitoring rule continuously creates events.
- Service Level Agreements are not met without prior warning.
- Applications run into failure without prior warning.
- Adaptation event does not lead to changes in service platform instance
- Adaptation event lead to unwanted changes in service platform instance
- Monitoring of events consumes too much of computational power

All these situations need through analysis; the reasons can be classified as follows:

- The platform architecture or implementation does not allow meeting the goals.

In this case a change request must be submitted to the Change Control Board (see definition in [D3.2] chapter 5.2.5). The problem can only be solved by changing the platform architecture, changing selected services, or adding new services.

- The adaptation rule does not solve the issue behind the triggering monitoring event.
- The monitoring rule does not meet its goals because
  - the threshold value is wrong / inappropriate
  - data for monitoring is not available
  - configuration of monitoring in service platform instance is wrong
  - mapping or translation of platform monitoring events to INDENICA event model fails
  - errors in the configuration of monitoring engines prevents from efficient monitoring of events

In these cases the improvement cycle for monitoring and adaptation rules is triggered.

In analogy to the Change Control Board, also a here a group of experts has to be named and empowered to perform the improvement cycle. Members of such a Board should be:

- The Platform Architect
- The Platform Variant Creator
- The Platform Integrator
- The Application Developer
- The Platform Operator

The improvement cycle phases are defined in analogy to the SGVM.

Phase: Plan

In this phase deviations and triggering situations are identified and analysed. The decision to submit a change request or to handle the issue within the measurement improvement cycle is taken.

Following the urgency evaluations a prioritized list of changes to measurements, monitoring rules and adaptation rules is maintained and an implementation plan is defined.

All work is performed jointly by the above listed group of experts.

Phase: Define

In this phase the identified needs for improvement are implemented in new, changed or dropped measurement definitions.

The measurement definitions are discussed and agreed upon by the group of experts

Phase: Implement

New or changed monitoring and adaptation rules are implemented using the respective tools like monitoring rule editor and adaptation rule editor. The implementation and maintenance of rules should be under responsibility the Platform Integrator.

The new or changed rules are deployed to the monitoring and the adaptation engine.

Phase Monitor

The running platform is monitored based on new / changed rules, events are created the trigger adaptation rules.

The Platform Operator is responsible for this phase, for reporting deviations and failures to the group of experts. Here the improvement cycle starts again.

5.3 Adaptation of base platforms

As described in Section **Error! Reference source not found.**, the adaptation will be managed by decentralized adaptation engine. However in order to allow one tool to cope with different types of communication mechanisms for firing adaptations, a unification of these mechanisms should be in the scope of Work Package 4.

With regards to the adaptation interface we have decided to follow similar path as we do with the monitoring interfaces, which is the unification of the models and implementation of several communication mechanisms, i.e. HTTP-Service Calls, direct messaging via queues, SOAP, etc.

The work on derivation of standardized adaptation interface is scheduled for the second half of the project, that’s why in this section we describe tailored adaptation interfaces for each separate service platform we use in INDENICA.

In the Warehouse Management System the adaptation service will be provided as SOAP-based Web Service. The service will be able to launch two different types of storage strategies – FastGoodsIn and OptimizedStorage depending on the situation in the warehouse.

In the Yard Management System the adaptations are more complex and include providing an ‘expected result’ as a feedback, which in the future will help us to create self-adapted sub-system, which does not only fire some adaptation actions, but also reacts, as the adaptation does not result with expected outcome.

The adaptation strategy in Yard Management System has been depicted in Table 3.

The adaptation of the Remote Maintenance System is based on the tailoring of the base platform – Mobicents, and is described in the D5.2 [D5.2] in the section 4.3.3.

	Monitored Subject	Rules	adaptationAction	Expected Result
Occupancy rate of	Docks			
	Parking areas			
	Whole yard	if dock.load == 100% and waitingBay.load >= 80%, notification to operator		decrease of occupancy
Trucks	Waiting time	if a truck waitingtime is over 4h, reschedule it with higher priority over newer trucks	post on */adpt/prioritizeTruck* with **truckId=x**	this and other trucks will be rescheduled
	Time on yard	if timeonyard ==high, waitingtime ==high and dock.load == medium, change algorithm to dynReschedulingAlgo	post on */adpt/algo* with **scheduling=dyn|static**	major and dynamic reschedulings reducing waiting time of trucks
Jockeys	Idle time	if avg idle time null/high, decrease/increase jockey count	post on /adpt/jockey with **action=dec|inc**	idle time is improved (other jockeys are intended to work in the warehouse instead)
Error rate	scheduling collision	if a collision takes place, resolve it with dynamic algorithm	post on */adpt/reschedule* with **appId=x&scheduling=dyn**	
Other		if a dock is disabled, reschedule all appoints to different docks	post on */adpt/dock/{id}* with **state=disabled|enabled**	rescheduling in/excluding this dock

Table 3: Adaptation strategy for Yard Management System

6 Summary and future work

As one can see, there are several different tools that are currently under development and/or tailored to the INDENICA Tool Suite for Deployment, Monitoring and Controlling of Virtual Service Platforms. Currently the work is focused on development of single tools, but as the project continues, there will be more concentration on integration of these tools within a single framework.

The conceptual and development work is also in line with the activities of Use Case Work Package (WP 5), which helped to get additional requirements and allow bringing monitoring and adaptation interfaces to existence.

Passing behind the half-way of the project, there are already some visible results like the simulation of the Tool Suite and supporting monitoring tools, that can be used to disseminate the project in front of the public as well as identifying the new possibilities for the industrial partners, their customers and other entities.

The future work of WP4 includes finishing ongoing development activities as well as starting the ones that have not started yet. There are several action points that are taken into the general work plan for the next six month:

- Using IRENE methodology to elicit adaptation (and monitoring) rules from generated requirement models
- Continue iterative approach using VbMF modelling to automate derivation monitoring and adaptation rules for the runtime environment
- Integration of monitoring tools and monitoring interface with live service platforms
- Testing of monitoring tools in emulated Virtual Service Platforms environment
- Create implementation of SOA Governance Vitality Method for INDENICA VSP
- Improve monitoring engines with self-adaptive capabilities (Adaptive Monitoring)
- Finalize the development of monitoring and adaptation interfaces

A Appendix 1: EcoWare Usage Guide

A.1 Installation and Setup

The installation of EcoWare requires the installation of the Siena event bus, the SCA and Infrastructure sensors, the EcoWare core system, the EcoWare Persistency service, and the EcoWare Dashboard.

The installation of EcoWare requires the installation of the Siena event bus, the SCA and Infrastructure sensors, the EcoWare core system, the EcoWare Persistency service, and the EcoWare Dashboard.

Siena Event Bus

The Siena Event bus can be installed to any host. From now on we will refer to the host as SIENA_HOST. The administrator needs to download Siena version 1.5.5. TO run Siena it is sufficient to go to the Siena folder and run:
java siena.StartServer -port SIENA_PORT -log LOG_FILENAME
This will run Siena on a specific port and log all its activities to a specific log file.

Infrastructure Sensors

The Infrastructure Sensors are based on collectd and run on Linux machines, which can be downloaded from http://collectd.org/. The installation of collectd requires that the JAVA_HOME environment variable be appropriately set. Then the administrator should move to the collectd folder and launch:

```
> configure
> make
> make install
```

At this point the installation is complete and the administrator should proceed to configure the tool, which can be done editing the /opt/collectd/etc/collectd.conf file.

```
LoadPlugin java
```

Next, the administrator should uncomment the Java plugin configuration and complete it so that it is as below:

```
<Plugin "java">
 JVMArg "-verbose:jni"
 JVMArg "-Djava.class.path =
       /opt/collectd/share/collectd/java/infrastructureSensor.jar"
 LoadPlugin "sender.DataSensor"
 <Plugin "sender.DataSensor">
 # To be parsed by the plugin
 </Plugin>
</Plugin>
```

At this point the Administrator should take the project named InfrastructureSensor and export the infrastructureSensor.jar. This can be done directly from Eclipse and

the jar should contain the folders sender, META-INF, org, and siena. Once the jar has been created it should be copied to /opt/collectd/share/collectd/java/.

Next the Administrator should take the file config.xml from the InfrastructureSensor project and copy it /opt/collectd/var/lib/collectd. The config file should be edited to point to the Siena Event Bus. The file should be as follows:

```
<infrastructure>
 <sienaServer>tcp:SIENA_HOST:SIENA_PORT</sienaServer>
 <vm>
  <eventName>InfrastructureSnapshot</eventName>
  <sendingPeriod>period</sendingPeriod>
  <publicationID>infrastructure_ID</publicationID>
  <host>HOST_NAME</host>
  <plugin>
   <memory>true</memory>
   <cpu>true</cpu>
   <interface>true</interface>
   ...
  </plugin>
 </vm>
</infrastructure>
```

From here the administrator can change the period with which new host snapshots are created (sendingPeriod), the id with which the events are published to the Siena event bus, the name of the HOST being sensed (HOST_NAME), and the infrastructure dimensions we want the snapshot to contain (memory, cpu, interface, etc.)

At this point the setup is complete and the administrator can start collectd, which will commence sending events to the event bus. This can be done using the following instruction:

```
> /opt/collectd/sbin/collectd -f
```

The -f option means that collectd will run in the foreground and log its activities to the console.

SCA Sensors

SCA Sensors are activated as monitoring intents within the Frascati runtime environment. The administrator needs to download the monitoring intent project, configure it with the Siena event bus details, create the appropriate jar, and add it to the Frascati runtime. This automatically makes SCA sensing available to all the SCA composites that are deployed to Frascati. Configuration is achieved in line 13 of the MonitoringHandlerImpl class file, where the administrator must pass "tcp:SIENA_HOST:SIENA_PORT" to the SienaOutputAdapter constructor.

The administrator can activate the SCA sensor on any SCA Service or Reference by placing the SCA "requires" attribute to "MonitoringIntent" in the Service or Reference's definition in the composite file. The Monitoring Intent will automatically send events to the Siena Event Bus.

EcoWare Core

The EcoWare core is a Java application that can be run directly from within Eclipse using the class Starter.Java. The Starter reads an EcoWare config file called "config.xml". This file attaches EcoWare data processors to the EventBus so that the appropriate KPIs can be calculated, and inter-level correlations achieved. A simple example of a config.xml file will be shown in the next subsection. A complete presentation of the configuration syntax is provided with the source code.

EcoWare Persistency

EcoWare persistency is a Java application that can be run directly from within Eclipse using the class Starter.Java. The Starter reads a config file called "config.xml". This file binds a primary data source with a set of secondary ones. Every time a new datum is published to the event bus by the primary data source, the persistency tool associates new data collected for each secondary data source. A simple example of a config file will be shown in the next subsection. A complete presentation of the configuration syntax is provided with the source code.

EcoWare Dashboard

The EcoWare Dashboard is a Java application that can be run directly from within Eclipse using the class Starter.Java. The Starter reads a config file called "config.xml". This file subscribes a set of charting panels to data published by various event sources for live viewing and configures other panels for accessing data that has been persisted to the EcoWare persistency tool. A simple example of a config file will be shown in the next subsection. A complete presentation of the configuration syntax is provided with the source code.

A.2 Example

EcoWare Core

In this example we are using EcoWare to correlate the average response time of a SCA service or reference call with an infrastructural snapshot. The SCA service or reference is identified by the id it uses to publish its request and response events to the event bus. The result of the average response time calculation is published using the avgrt_ID. The interval unit and value identify the amount of past time that should be considered when calculating the average response time, and the output unit and value identify how much time passes between two subsequent outputs. The correlation is defined in the XML's aggregator node. The aggregation takes two events, a primary one and a secondary one. The primary one is the average response time identified by the avgrt_ID. The secondary one is the infrastructure snapshot identified by the infrastructure_ID. The result of the aggregation is published to the bus using id aggregated_ID. Every time a new primary event is seen on the bus, the aggregation correlates all the secondary events that were seen in a past temporal window defined by the "collection" node.

```
<ecoware>
   <sienaServer>tcp:SIENA_HOST:SIENA_PORT</sienaServer>
   <KPI>
      <name>AvgResponseTime</name>
      <subscriptID>SCA_publish_ID</subscriptID>
      <publicationID>avgrt_ID</publicationID>
      <computation>
         <intervalUnit></intervalUnit>
         <intervalValue></intervalValue>
         <outputUnit></outputUnit>
         <outputValue></outputValue>
      </computation>
   </KPI>
   <aggregator>
      <subscriptID1>avgrt_ID</subscriptID1>
      <subscriptID2>infrastructure_ID</subscriptID2>
      <publicationID>aggregated_ID</publicationID>
      <primaryEventName>AvgResponseTime</primaryEventName>
      <secondaryEventName>InfrastructureSnapshot</secondaryEventName>
      <collection>
         <intervalUnit></intervalUnit>
         <intervalValue></intervalValue>
      </collection>
   </aggregator>
</ecoware>
```

EcoWare Persistency

In this simple example we persist the average response time and the infrastructural data for future use. The configuration contains the same information from the above example, except the collected data are no longer re-published together to the event bus but persisted to the persistency database.

```
<persist>
   <sienaServer>tcp:SIENA_HOST:SIENA_PORT</sienaServer>
   <aggregate>
      <KPI>
         <subscriptID>avgrt_ID</subscriptID>
         <kpiName>AvgResponseTime</kpiName>
      </KPI>
      <secondary>
         <subscriptID>infrastructure_ID</subscriptID>
         <secondaryEventName>
            InfrastructureSnapshot
         </secondaryEventName>
   <collection>
            <intervalUnit></intervalUnit>
            <intervalValue></intervalValue>
         </collection>
      </secondary>
   </aggregate>
</ persist >
```

EcoWare Dashboard

In this simple example we create a view of the average response time and of its aggregated infrastructure data. Since each infrastructure snapshot contains multiple dimensions (CPU, memory, etc.) we need to identify which of these information we want to show

```
<dashboard>
 <sienaServer>tcp:SIENA_HOST:SIENA_PORT</sienaServer>
 <aggregate>
   <frameTitle>Average Response Time and correlated data</frameTitle>
   <SubscriptID>avgrt_ID</SubscriptID>
   <Name>AvgerageResponseTime</Name>
   <ValueAttribute>value</ValueAttribute>
   <Cutoff>cutoffValue</Cutoff>
   <ChartTitle>Average Response Time</ChartTitle>
   <secondaryEvent>
      <subscriptID>infrastructure_ID</subscriptID>
      <name>InfrastructureSnapshot</name>
      <valueAttribute>cpu_0_cpu_user.value</valueAttribute>
      <valueCutoff>cutoffValue</valueCutoff>
      <collection>
            <intervalUnit></intervalUnit>
            <intervalValue></intervalValue>
         </collection>
      <chartTitle>User CPU</chartTitle>
   </secondaryEvent>
 </aggregate>
</dashboard>
```

In this case we are creating a panel that shows the average response time with the corresponding user cpu usage. Each new average response time is associated with the past cpu snapshots collected within the collection range defined in the “collection” node. The configuration also has cutoff values that can be shown on the value plots as horizontal lines. This makes it easier for the administrator to establish when the values being plotted are reaching noticeable amounts. Finally, it also contains some extra information regarding the titles of the various data that are to be shown.

B Appendix 2: SPASS-meter Quick Guide

This is a brief introduction into the usage of the SPASS-meter instrumentation framework. As the support for monitoring non-Java SUMs is currently under development, we will discuss here exclusively the installation, the setup and an example for Java SUMs.

B.1 Installation

SPASS-meter is packaged into a set of Java archives (JAR) for Windows and Linux operating systems as they differ in the included native data gatherer library. Depending on the instrumentation mode, the JARs are linked to the SUM in different ways:

- **Static instrumentation:** The `SPASS-meter-ant.jar` contains the instrumentation layer, the static instrumentation tool, and the integration into the build process. Currently, a simple ANT task for build-process integration is provided. For runtime, the `SPASS-meter-static.jar` includes the probe collection layer, the data aggregation layer, the data presentation layer as well as prepackaged extensions to such as the implementation of the INDENICA monitoring interface for integration with the INDENICA runtime environment. The `SPASS-meter-static.jar` needs to be included into the class path of the SUM.
- **Dynamic or mixed-mode instrumentation:** The `SPASS-meter-ia.jar` contains the instrumentation layer and the Java instrumentation agent. Due to technical reasons, the Java instrumentation agent loads dynamically two further JARs, one for boot time and one for runtime. The `SPASS-meter-boot.jar` contains annotations and interfaces which need to be present at boot time of the SUM in order to resolve dependencies inserted into the SUM or (dynamically) into the Java library. The `SPASS-meter-rt.jar` contains the upper layers for processing notification calls as well as the analysis extensions similar to `SPASS-meter-static.jar` described above.

If the SPASS-meter monitoring scope specification is given in terms of source code annotations, the `SPASS-meter-annotations.jar` needs to be included into the class path at development time.

In order to install SPASS-meter the JARs mentioned above just need to be copied into one directory (lib directory). The specific JARs are specified as JVM parameters, either the java agent parameter or the class path parameter. In the dynamic case, the location of JARs which are loaded at runtime is inferred based on the already specified JARs.

B.2 Setup

For applying SPASS-meter to a SUM, the monitoring framework needs to be configured. The configuration is twofold: a global configuration which determines the basic operation mode and defaults as well as the monitoring scope specification. The global configuration is given as part of the JVM or tool parameters, respectively.

The monitoring scope can either be specified using source code annotations, e.g. in handcrafted or generated code, or as an XML file. The monitoring scope defines the monitoring groups, i.e. the relevant classes (and methods if required) as well as the individual resources to be monitored. Depending on the specified analysis extensions, the results of monitoring may be a summary file, live events etc.

In INDENICA, the XML monitoring scope specification will be generated from the variability model (runtime variabilities) and further input taken from the monitoring requirements or specification (WP2/WP3). As output, monitoring events will be sent over the monitoring interface event mechanism to the INDENICA runtime environment. Required parameters for the event mechanism will be taken from the deployment specification.

B.3 Example

The example below illustrates the generic application of SPASS-meter to Mobicents JAIN SLEE [MJSLEE]. As the Mobicents source code shall not be modified for monitoring its services, the following XML monitoring scope specification defines that

- Monitoring starts with the startup of the Mobicents SLEE container
- All resources supported by SPASS-meter except for memory usage (`memAccounting` mode `CREATION`) are accounted for all dynamically started Mobicent services (implementing `javax.slee.Sbb`)
- Monitoring stops at the end of the Mobicents SLEE container.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://sse.uni-hildesheim.de/instrumentation"
  memAccounting="CREATION">

  <namespace name="" typeOf="javax.slee.Sbb"/>

  <namespace name="org">
    <namespace name="mobicents">
      <namespace name="slee">
        <namespace name="container">
          <module name="SleeContainer">
            <behavior signature="initSlee()">
              <startSystem/>
            </behavior>
            <behavior signature="shutdownSlee()">
              <endSystem/>
            </behavior>
          </module>
```

```
        </namespace>
      </namespace>
    </namespace>
  </namespace>
</configuration>
```

For dynamic instrumentation the Java agent is given as a JVM parameter. Below, the Java agent is packaged in `spass-meter-ia.jar`, the monitoring scope is given in the `mobicents.xml` file and regular update events are issued each 2000 ms.

```
  -javaagent:instrumenter\spass-meter-
ia.jar=xmlconfig=mobicents.xml,
    outInterval=2000
```

The static instrumentation of Mobicents is illustrated below in terms of a simple ANT task (fragment). The static instrumentation tool takes two directories with JAR files as input, produces instrumented jar files as output and considers the specified options currently specified in the same format as for dynamic instrumentation shown above.

```
  <spassInstrumenter
     classpathref="classpath"
     in=”lib/*.jar, server/default/deploy/mobicents-slee/lib/*.jar”
     out="instrumented"
     params="xmlconfig=mobicents.xml,outInterval=2000” />
```

C Appendix 3: Indenica Runtime Platform Demonstrator

This section presents a guide on the steps necessary to set up the INDENICA runtime platform in Eclipse, as well as a standalone application.

C.1 Initial Eclipse Project Setup

Getting the project up and running in Eclipse for the first time involves the following steps:

- Don't worry about errors due to missing build.properties or due to an unbound classpath variable (e.g. M2_REPO), as they should be fixed by maven automatically.
- Set your default installed JRE to your JDK (required for maven)
- Install a maven integration, e.g. m2eclipse
- Run maven on the project in order to populate the maven repository
- Refresh your project. Additionally, it might be needed to
    - create an empty `build.properties` file
    - modify your `eclipse.ini` to point your -vm to the JDK-VM to get rid of a tools:jar:1.5.0 error

C.2 Getting Project Dependencies

The runtime platform uses MongoDB and RabbitMQ for data storage and messaging. These components must be running in order for the platform to work.

You can either install these requirements on your development machines or use a prepared virtual instance.

C.2.1 Manual Installation of Dependencies

For installation of MongoDB and RabbitMQ, please refer to the respective project web sites. After installing and starting the components, you will need to adjust the platform environment configuration to point to your local instances.

NOTE: At the moment, this information is scattered throughout several files. Future refactoring and cleanup will improve this situation. Using the preconfigured virtual machine described below should be an easier way to get the necessary dependencies running.

Currently, environment configuration is stored in the following files in src/main/resources/

- src/main/resources/META-INF/sca-deployables/*.composite: The *.composite files contain information about how to instantiate runtime instances. The RepositoryComponent section contains a reference to the MongoDB instance. Adjust the dbAddress property accordingly.

- src/main/resources/files2DB/properties/*.properties: The *.properties files contain configuration information for the initial data population module. The eventRepositoryAddr property should point to the RabbitMQ instance. Adjust this property accordingly.

C.2.2 Use a preconfigured Virtual Machine with all Dependencies

The current default configuration assumes that there are active RabbitMQ and MongoDB instances running on host `192.168.56.101`.

These requirements can be easily fulfilled by perusing `vagrant` and a virtual machine provided by TUV. The steps necessary to start the virtual machine are:

- Install vagrant (see http://vagrantup.com/)
  ```
  gem install vagrant
  ```
- Add the `indenica_support_dependencies` base box:
  ```
  vagrant box add indenica_support_dependencies \
      http://db.tt/qQcjgPzp
  ```
- Initialize the machine:
  ```
  mkdir support_components && cd support_components
  vagrant init indenica_support_dependencies
  ```

You only have to perform these steps once. After the initial setup, the configured virtual machine can be started using:

```
vagrant up
```

After a few minutes, the virtual machine should be running, the network settings applied and the environment ready to go.

The virtual machine can be stopped using:

```
vagrant halt
```

The machine can be started again using vagrant up. For further information, please refer to the Vagrant Documentation.

C.3 Starting the Platform

C.3.1 Command Line

When you have the dependencies running, you need to deploy the platform configuration data before running the simulation:

mvn exec:java -Dexec.mainClass="indenica.deployment.utils.Populator"

With all prerequisites deployed, the Warehouse simulation can be started using:

```
mvn exec:java -Dexec.mainClass="indenica.deployment.Launcher" \
              -Dexec.arguments="UseCaseWP4Runtime"
```

This will start the demo you saw at the Munich meeting.

C.3.2 Eclipse

To start the simulation from within Eclipse, run the indenica.deployment.usecase.UseCaseLauncher class as a Java Application.

To stop the runtime instance, hit return in the console.

References

[BH04] W. Binder and J. Hulaas. A Portable CPU-Management Framework for Java. *IEEE Internet Computing*, 8:74–83, September 2004.

[CO04] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information. *SIGSOFT Softw. Eng. Notes*, 29:1–4, September 2004.

[SGF 2009] The Open Group: The Open Group SOA Governance Framework; Draft Technical Standard, 2009. www.opengroup.org/projects/soa-governance.

[D1.2.1] INDENICA Deliverable D1.2.1 - Requirements Engineering Framework, Language and Tools for Service Platforms (Interim), 2011-10-31

[D2.3.1] INDENICA Deliverable D2.3.1 - Service Platform Infrastructure Repository Concept & Realization (Interim), 2012-01-31

[D3.1] INDENICA Deliverable D3.1 – View-based Design Time and Runtime Architecture for Tailoring VSPs, 2011-10-18

[D3.2] INDENICA Deliverable D3.2 - Architecture for Role-Based Governance of Virtual Service Platforms, 2012-01-31

[D5.2] INDENICA Deliverable D5.2 - Report on Concepts for Tailoring and Extending Service Platforms, not yet published

[JI12] JXInsight/OpenCore, 2012, jinspired.com/.

[JSON] Internet Engineering Task Force, RFC 4627, 2006, http://www.ietf.org/rfc/rfc4627.txt

[MJSLEE] Mobicents JAIN SLEE, Red Hat Middleware LLC, 2008, http://www.mobicents.org/slee/intro.html

[OSOA] Open SOA (2007). Service Component Architecture (SCA) Specifications V1.00. http://www.osoa.org

[O08] Oracle, Java Management Extensions (JMX) Technology, 2008, www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[OW2C10] OW2 Consortium, WildCAT, 2010, wildcat.ow2.org/

[WSCoL] L. Baresi, S. Guinea, “Self-supervising BPEL Processes,” IEEE Transactions on Software Engineering

[XML] W3C, Extensible Markup Language, 2008, http://www.w3.org/TR/REC-xml/