Engineering Virtual Domain-Specific Service Platforms

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

Report Describing a Framework for Deployment, Monitoring & Controlling of Virtual Service Platforms

Abstract

This document describes the Framework for (semi-)automatic instantiation and supervision of Virtual Service Platforms as a part of INDENICA.

The document includes the description of all parts of the framework including Monitoring Engine, Adaptation Engine, Repository, Deployment Manager and Integration Interface. This report includes requirements and the architectural description of the components and shows how these elements work together with regards to the outcomes of other Work Packages.

Document ID:	INDENICA – D4.1
Deliverable Number:	D4.1
Work Package:	4
Type:	Deliverable
Dissemination Level:	PU
Status:	draft
Version:	0.7
Date:	2011-09-30
Contributing Partners:	PDM, SAP, SIE, TUV, SUH, UNIVIE, TEL

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

Version History

0.1	30 May 2011	initial TOC version
0.2	07 July 2011	Initial TOC version revised
0.3	31 August 2011	TOC updated and finalized
0.4	16 September 2011	First integrated version
0.5	19 September 2011	Polishing and integration
0.6	26 September 2011	Additional content added
0.7	30 September 2011	Final version - polished and integrated

Document Properties

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

Table of Contents

1 Introduction

This work package - integrates the whole technical work in the project into one single framework, which will allow deployment and runtime governance of Virtual Service Platforms. It provides development time, compile time, and runtime modules for managing the deployment of tailored service platform components, services, and applications in heterogeneous environments. The focus is put on the development of tools for supporting the (semi-) automatic instantiation and runtime governance of Virtual Service Platforms, which are using tools and methodologies provided by WP1, WP2, and WP3.

WP4 will provide a complete framework and methodologies to manage the lifecycle of Virtual Domain-specific Service Platforms including deployment, runtime governance and development of supporting tools and technologies.

This document describes the concept of such framework, its functionalities and the way it will work including used tools, components, and environments. The designed framework will be described and the concepts will be implemented in the next two years of the project in two phases (interim and final) and provided with the documentation included in deliverables 4.2.1 and 4.2.2 respectively.

Special attention is brought to present novel approaches for Runtime Adaptive Controllers, which are responsible for assuring end-to-end QoS by dynamic changes of runtime environment on both service and platform level.

This document consists of following sections: in section 2 there is an overall view of the Framework including high-level diagrams. In Sections 3, 4, and 5 there are descriptions of Framework's main components responsible for Monitoring, Adaptation, and Deployment respectively. Integration and cooperation with different work packages is described in section 6. In section 7 we present conclusion of the current work in this work package and we discuss briefly future steps.

2 Overall description of the Framework

The described framework is responsible for managing the deployment process and runtime monitoring of the Virtual Service Platform (VSP).

Heterogeneity of the whole environment requires the integration of different domain-specific platforms on service level which is supported by tools developed in WP1, WP2 and WP3. This integration process gives an ability to use a common interface and messaging technology for monitoring and adaptation purposes.

The runtime monitoring and adaptation framework will be equipped with self-managing capabilities, which means that it will be able to preserve in dynamic environment by adjusting to the current situation. It is achieved by additional runtime modules which support dynamic adaptations as well as runtime monitoring to meet end-to-end QoS requirements.

The proposed approach to the implementation of the framework for runtime monitoring and adaptation of VSP is based on the Service Component Architecture (SCA) model. As a result we are able to provide a programming model for building applications and system based on Service Oriented Architecture, which takes in account the domain-specific requirements for both platform and service and the target deployment infrastructure.

We have decided to base the WP4 runtime environment on SCA because it provides a "concise and flexible model for describing and developing SOA applications" [1]. Among the benefits of using SCA [1] are: "rapid development and increase in productivity, higher organizational agility and flexibility, return on Investment through reuse". Moreover, SCA is a mature concept with a number of reliable open source platforms which support it, such as Apache Tuscany[1] or Fabric3[2].

The proposed framework will be declarative in terms of accommodation domain-specific variability in non-functional requirements of both platform and application components. The proposed framework includes also design-level constraints into compile time bindings of features to services which appropriately take into account the target components. Moreover, domain-specific non-functional requirements will be automatically compiled into runtime policies to be used for dynamic instantiation or migration of components.

To achieve the objective of the framework, the following actions will need to be done:

- Development of deployments manager for the deployment of service platforms and the virtual service platform
- Development of runtime modules to support dynamic adaptations and runtime monitoring to meet end-to-end QoS requirements

[1] http://tuscany.apache.org/home.html

[2] http://www.fabric3.org/

- Development of common monitoring and adaptation interfaces for the heterogeneous environment
- Development of runtime policy-driven adaptive components which will interface with platform infrastructure and application components to manage dynamic instantiation and migration to meet QoS requirements

In summary, the proposed monitoring and adaptation framework significantly improves the lifecycle management of Virtual Service Platforms. An overall notion of the WP4 framework is presented in the Error! Reference source not found..

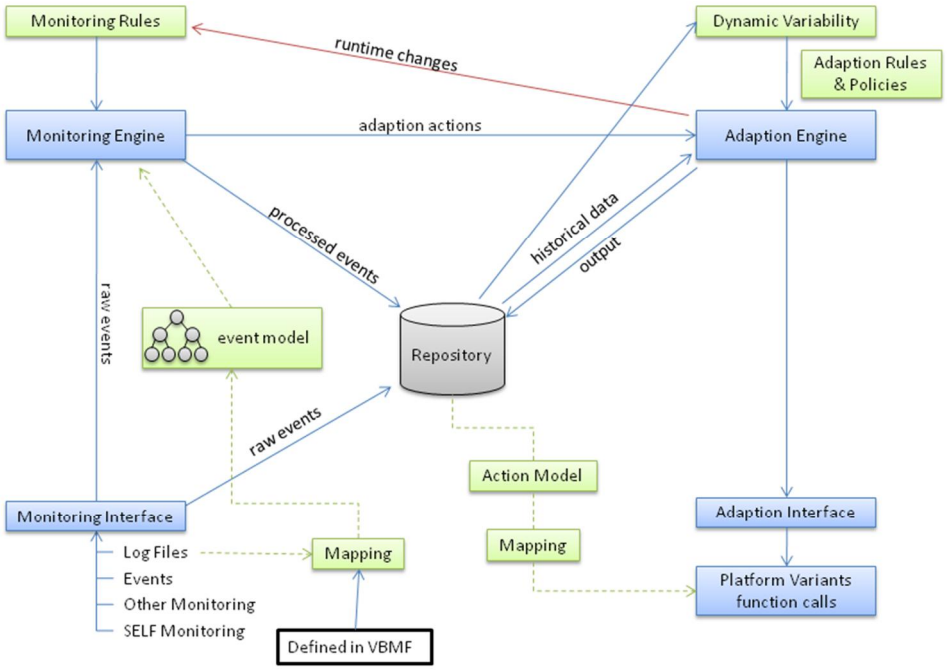

Figure 1: Overall architecture of WP4 Framework

Components that will be developed within WP4 are marked in blue colour, while supporting models and model instances are marked in green. Cooperation between the components and usage of models are visualized as directed arrows.

The central element of the figure is the Repository which acts both as a back-end database and online caching storage. From the monitoring perspective, the Repository is used to store raw events directly from the monitoring interface (from domain-specific platforms) as well as processed events in order to use them both for adaptation and presentation layer (dashboard). The Monitoring Engine uses monitoring rules to catch proper events or sequences of events. The Repository also stores models which define how the platform variants differ from each other, i.e. particularly the partially instantiated variability model with unbound runtime variabilities and the related asset model which points to the concrete components realizing the options and alternatives represented by the variabilities.

Based on the monitored events, the Adaptation Engine launches adaptation actions accordingly to previously defined adaptation Rules and Policies. In some cases an adaptation action may require a change in how the Monitoring Engine monitors different components (red arrow). Adaptation Engine uses the adaptation interface to perform adaptations directly on domain-specific service platforms.

In order to make it possible to use common monitoring and adaptation interface for arbitrary underlying service platforms, these platforms will need to comply with these interfaces.

2.1 Monitoring Engine

The main purpose of monitoring is to filter and analyze events in various ways, and to respond to events of interest in real-time. The monitoring facilities are also responsible for efficiently gathering required information from the VSP components. This includes determining optimal strategies for monitoring, in order to reduce overhead and provide 'detailed enough' information. Furthermore the Monitoring Engine will track QoS and SLA specific parameters from domain-specific platforms as well as components of the Virtual Platform. The Monitoring Engine uses monitoring rules provided by the Adaptation and Monitoring Rule Editor. These Rules are generated based on QoS parameters and SLA.

The input data for the Monitoring Engine will be both events generated by the execution on the service platforms and instructions from Deployment Manager.

The monitoring facilities store the monitoring data, reports and logs in the Repository. Results from the Monitoring Engine are sent to Adaptation Engine and dashboard/user interface.

As a part of Monitoring Engine complex event processing (CEP) is used to deliver high-speed processing of many events across all the layers. Existing technologies used to develop the Monitoring Engine will include ESPER, JMX and ActiveMQ.

2.2 Adaptation Engine

The Adaptation Engine is responsible for adapting components according to specified policies and rules, e.g., as reaction to changes in the environment. The complexity of the envisioned target platforms makes it necessary to add autonomic adaptation capabilities controlled by policies and high-level objectives. The Adaptation Engine receives input from the Monitoring Engine and performs reasoning based on that data. Additional information affecting the behaviour of the Adaptation Engine, such as adaptation policies and system capabilities, are stored in the Repository. Furthermore, the Adaptation Engine is able to refine and optimize provided policies based on observations and experiences gathered after applying them. There are various kinds of adaptation actions which can be triggered by the Adaptation Engine, ranging from direct adaptation of a concrete platform to exchanging models at runtime.

2.3 Repository

The Repository acts as a database for input and output of monitoring and adaptation engines. The Repository also stores the information about on-going adaptation activities in order to enable evaluation of these activities. Moreover, the Repository is also used for storing monitoring data and policies concerning constraint violations. Monitoring data stored in the Repository are pre-processed for further evaluation. The Repository is used both for support both of the development and runtime environment. More details about the Repository are provided in D2.3.1.

2.4 Integration Interface

INDENICA provides a common Interface (Figure 2) which enables various platform providers to connect to the INDENICA platform. This interface enables seamless integration of platforms with INDENICA Virtual Service Platform. The main task of this interface is to enable the exchange of control and adaptation instructions between the monitoring and adaptation engines in order to communicate with the domain-specific platforms and vice-versa.

The Monitoring Interface mediates between Monitoring Engine and external service or platform. It is based on an event-based information exchange model with a standardized event message format. The events can be hierarchised based on the "level" of the publishing entity (service platform).

The Adaptation Interface mediates between the INDENICA Adaptation Engine and the external platform adaptation services. The message format used in this interface is based on a standardized, formal language which will be developed as one of the framework components.

The Integration Interface is described in section 3.2 (monitoring) and 4.4 (adaptation).

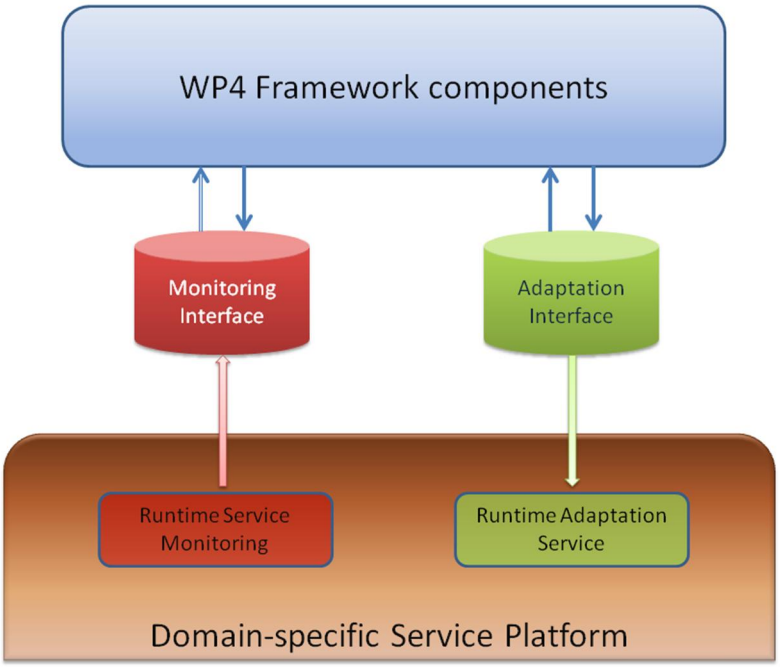

Figure 2: Integration Interface

2.5 Deployment of WP4 components

The proposed model for the INDENICA framework is based on the SCA methodology which uses a wide range of technologies for service components and for the access methods which are used to connect them. One feature of the proposed model is that it is also possible to automatically control the deployment process of underlying SPs.

Based on preferences of consortium members the Apache Tuscany platform has been chosen as a base runtime environment for WP4. Apache Tuscany provides a comprehensive infrastructure for SOA development and management that is based on the Service Component Architecture (SCA) standard. It "extends the SCA programming model with its support for many different binding types, implementation types, and runtime environments" [2]. Apache Tuscany currently offers a stable implementation of SCA specification version 1.0, while an implementation of version 2.0 is currently available as beta.

Deployment components from one side will be responsible for applying output from compile time tools (models, generators from different WPs) and from the other one to run policy-driven adaptive components to support dynamic instantiation and migration of underlying components.

Deployment of the whole WP4 environment will be partially automated and users or the administrator will use configuration files for manual inputs.

Deployments process will also take into account self-monitoring which will enhance the governance of the Virtual Platform at runtime. Such monitoring will be able to provide information about various problems of WP4 components that might occur at any time.

3 Monitoring of Virtual Service Platforms

3.1 Monitoring Overview

The main purpose of monitoring is to filter and analyse runtime events in various ways, and respond to of interest in real-time. The monitored events can originate either from service platform instances or from the running Virtual Service Platform (VSP) instance. The purpose of runtime monitoring is to ensure the fulfilment of functional and non-functional requirements imposed on the running VSP instance by providing feedback to the Adaptation Engine. Additionally monitoring information will be provided in human readable form to a System Operator who can also interact with the system as required. Furthermore, monitoring is a prerequisite of adaptation at runtime, as discussed in Section 4.

Monitoring will be implemented in the Monitoring Engine runtime component. This component will rely on Complex Event Processing (CEP) to deliver high-speed processing of many events across all the layers. One of the key features of Complex Event Processing is the capability to "continuously process event streams to detect a specified confluence of events, and trigger a specific action when the events occur" [3]. As a result of applying CEP, the platform can be monitored in near-real time based on the flow of events (see [4] for examples). Thanks to applying CEP, it will be possible to process and aggregate events received by the Monitoring Engine into complex events based on predefined functional and non-functional requirements.

The interactions of the Monitoring Engine with the underlying Service Platforms and other components of the VSP are illustrated in Figure 3. The basic assumption is that the Monitoring Engine should be able to receive and understand events from any current or future underlying service platform. In order to ensure this, a generic and extensible Monitoring Event Model is introduced (described in section 3.2), which is used to convert events generated by SPs to internal INDENICA events. The implementation of this conversion will take place in the Monitoring Interface component, which is described in section 0.

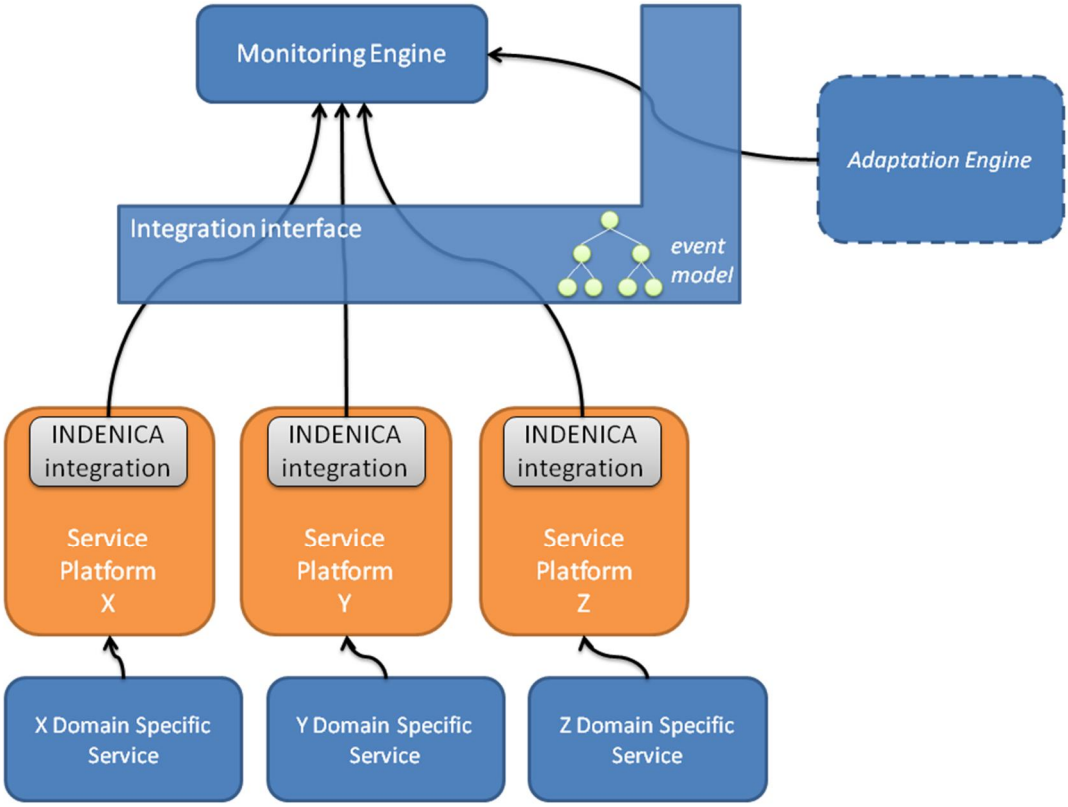

Figure 3: Interaction of the Monitoring Engine with the underlying Service Platforms and the VSP

As can be seen in Figure 3 we assume that each platform will have to be adapted to INDENICA monitoring mechanisms, which is denoted by the 'INDENICA Integration' box. To comply with INDENICA monitoring, each service platform will have to be extended by a module which will be responsible for establishing a monitoring communication channel between the service platform and the INDENICA monitoring interface. Whether this module could be generated automatically by INDENICA tools is still an open issue, but our goal will be to automate this integration as much as possible.

What can also be observed on Figure 3 is that the Monitoring Engine will be responsible for self-monitoring of the runtime platform, by handling monitoring events from the Adaptation Engine. The events generated by the Domain Specific Service instance will have to comply with the event model, as the service will be subject to the same monitoring and adaptation workflow as the underlying Service Platforms.

We identify two key challenges for monitoring implementation in INDENICA. The first is to provide a solution for associating a shared semantics to various events across diverse service platforms in order to support monitoring of arbitrary service platforms. The second is to derive universal monitoring interfaces (and an approach to their implementation) able to support arbitrary service platforms. The following sections describe the INDENICA design for solving these challenges.

3.2 *Monitoring Event Model*

The purpose of the monitoring event model is to provide a structure for events that should be understood and processed by the INDENICA framework. This structure should be generic enough to cover all events from any existing service platform. At the same time it should be detailed enough to allow for creation of detailed rules for triggering adaptation actions under specific conditions. Our general approach is twofold: on the one hand, we provide an event hierarchy, which serves as the main classifier for concrete events. This main event model defines what types of events exist in the system. In addition to the main event model, and orthogonal to it, there is an event feature model. This model defines what additional purposes any event can serve. More precisely, any concrete event is of exactly one type from the main event hierarchy, but may implement any number of features from the event feature model. The event model presented here is loosely based on earlier work [6]Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found. that some of the INDENICA partners have carried out as part of the VRESCo project [7].

Our initial version of the main event model is depicted in Figure 4. All events inherit from the common BaseEvent, which contains some basic event information (timestamp of when the event has been triggered, unique event identifier, and some optional additional data). On the second level of hierarchy, we distinguish four broad classes of events. ManagementEvents refer to events triggered by internal VSP activities, such as querying the Repository or triggering an adaptation. Similarly, LifecycleEvents are triggered whenever the status of one of the VSP components changes, for instance, if a new domain-specific service is started, finished starting, or is stopped. ExecutionEvents tracks service invocations in the VSP. There are events triggered when invocations start, finish, fail or are interrupted. Finally, StatusEvents are triggered by sensor-style components. These events are typically not associated with any concrete activity in the VSP or the platforms. Instead, StatusEvents are often generated periodically.

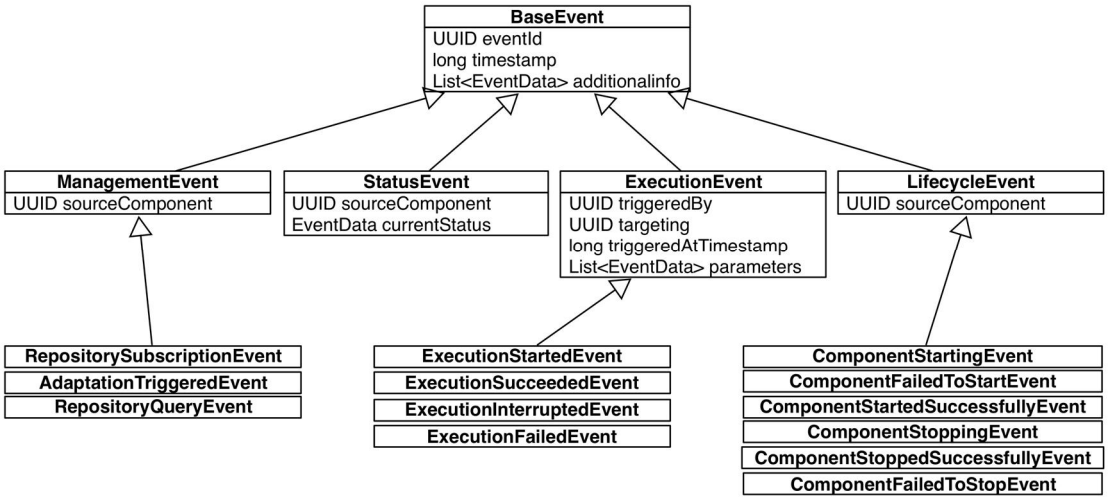

Figure 4 – Main Event Model

The third hierarchical level is formed by concrete events from the event classes discussed above. We have exemplified some important events from the ManagementEvent, ExecutionEvent and LifecycleEvent classes, even though this model is not necessarily complete. In the remainder of the project, we plan to extend the model with additional event types and classes, as we identify them. Note that there is no further concretization for StatusEvents at the moment.

Figure 5 depicts the initial version of the event feature model. Like the main event hierarchy, this model is not exhaustive, i.e., it is to be expected that we will identify additional features that have to be added to the model in the course of the project. Unlike the main event hierarchy, the feature model is mostly a flat structure. Any concrete event may implement any number of features from the feature model. In its current version, we provide features for role-based access control (RBACEnabledEvent, which contains the RBAC typical data like subject, role and context), versioning (VersioningAwareEvent, which indicates that an event knows which version of a component has produced it), models at runtime (ModelAwareEvent and ModelInstanceAwareEvent, linking events to concrete models and model instances from the view-based modelling framework), and aggregation (AggregateEvent, which indicates if an event has been produced from a series of lower-level events using techniques of complex event processing).

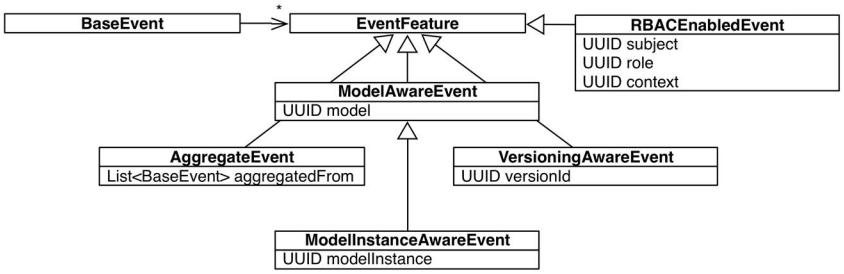

Figure 5 - Event Feature Model

We now demonstrate an example of the usage of the INDENICA event model based on a concrete event from the case study. Let us consider an initiation of a video call between two system users. The main concrete event named VideoCallInitiatedEvent is fired by invocation of a domain-specific service at one of the underlying platforms. Because of that, this event is of the type ExecutionStartedEvent. The event also implements the event features ModelInstanceAwareEvent and RBACEnabledEvent. The resulting event including all event properties is depicted in Figure 6. Note that some of the properties are fixed for certain types of events, for instance the property *UUID targeting* always refers to the UUID of the domain-specific video-on-demand service. Finally, the VideoCallInitiatedEvent also contains a larger number of event-specific fields, which are specific to the video-on-demand domain. The INDENICA event model allows events to contain arbitrary additional information, which may be ignored by generic monitoring rules, but which may be important to some components, which are aware of the specifics of the respective domain.

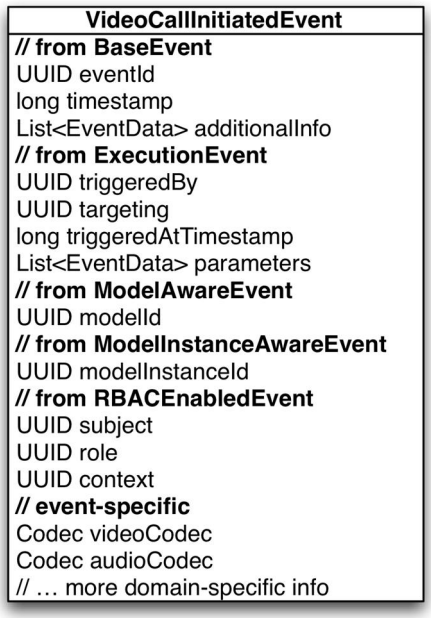

Figure 6 - Example Event

3.3 Monitoring Engine

The Monitoring Engine (Figure 7) can be best described as a flexible runtime validation platform that exploits a Publish/Subscribe middleware to distribute information to different analysers. Though our initial architecture presented within this document consists of only two analyzers (described below) the architecture will be able to support multiple event analysers which will be added at later stages if required.

One of the event analyzers will be implemented with the use of the Esper Complex Event Processing Engine to process collected events and to provide inputs to other analysers if needed. Esper provides an Event Processing Language (EPL) that allows developers to easily define complex event conditions, correlations, and aggregations, thus effectively minimizing the effort required to keep track of a distributed system's behaviour.

The second analyzer will be implemented based on a rule engine (such as e.g., the JBoss Rule Engine) used to evaluate business-related rules and which could be used by other external analysers for special-purpose validations. The JBoss Rule Engine (Drools) is a business rule management system (BRMS). It supports the JSR[3] standard for its business rule engine and enterprise framework for the construction, maintenance, and enforcement of business policies in an organization, application, or service.

The publish/subscribe event notification service ensures the correct distribution of information with a large number of communicants and high volumes of events. It will also be possible to dynamically combine multiple analyzers in a pipe-and-filter fashion.

[3] http://jcp.org/en/jsr/detail?id=94

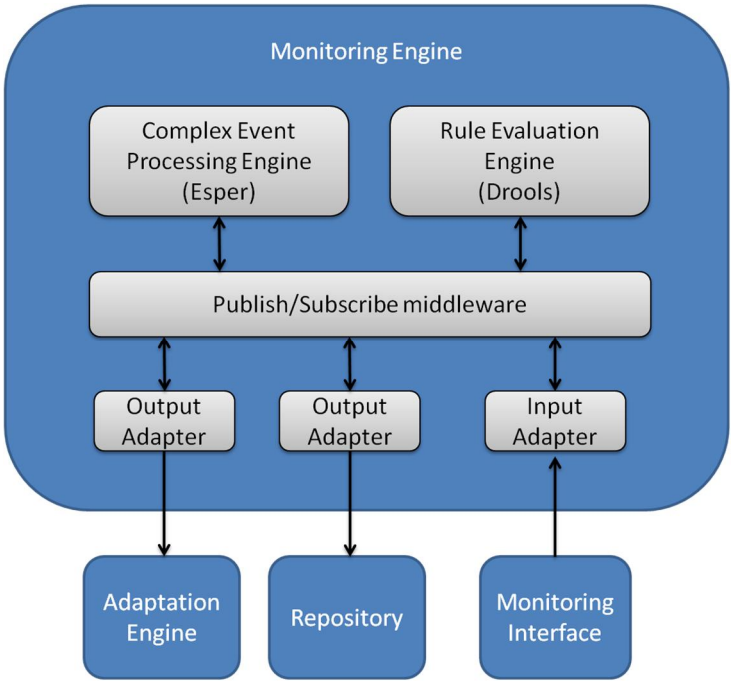

Figure 7: The architecture of the Monitoring Engine

The middleware also defines a normalized event format for the data that flow through P/S infrastructure, and provides configurable adapters for the different components.

3.3.1 Monitoring rules

Monitoring rules, to feed the different analysers, can be extracted semi-automatically from the goal model (see D1.2.1 for more details on goals). Soft goals, which are dedicated to non functional (or qualities of service) requirements can easily be used to identify the quality dimensions of interest, and if, properly formalized, they can also be used to derive monitoring directives directly.

There are a number of approaches that can be used to specify goals. Currently we identify the following existing possibilities:

- Natural language. If goals are only rendered in natural language, the monitoring directives can only be generated by hand.
- Formalized language. If goals are specified using languages like OCL (Object Constraint Language) or even more sophisticated languages (e.g., Linear Temporal Logic), we could easily apply transformation techniques based on the two different meta-models, to translate constraints in the problem space into directives into the solution space.

The decision on a specific method of specifying goals will be made during the implementation phase of the project.

Any monitoring activity requires the availability of suitable data. Its proper collection depends on the probes available in the infrastructure. Probes mainly differ on how (push/pull mode), and when (periodically/when certain events take place) data must be collected. If data are collected in a push mode, the Monitoring Engine just

receives them from the corresponding probes. Instead, if data are collected in a pull mode, the infrastructure is responsible for activating the collection (periodically or at specific execution points). Every time a new datum is available the middleware distributes it to all interested parties.

The definition of the monitoring directives depends on the monitoring capabilities provided by the Monitoring Engine. There are a several possible types of constraints that could be verified by the engine (untimed or temporal, fuzzy or crisp). There are also two different approaches to implementing verification of these constraints, that is synchronous(if a result is provided right after the constraint is verified) or asynchronous (if a result is provided in a different moment than when the constraint is verified).

In general this section presented a number of aspects related to the implementation of monitoring rules, together with a number of possible solutions to each of these aspects. All the presented options will be evaluated during implementation phase of the project to achieve the desired flexibility of the Monitoring Engine.

3.4 Monitoring Interface

INDENICA will create an integration interface for monitoring which will enable various platform providers to comply to and to enable seamless integration with the Virtual Service Platform. The integration interface for monitoring will be event-driven and will rely on a standardized event message format. Each platform provider will be responsible for defining a mapping between the events generated by their platform and selected events from the monitoring events model of the INDENICA platform. This mapping will be all defined using INDENICA design-time tools, which will generate pieces of code that will facilitate the integration of the service platform with INDENICA.

The monitoring interface is an entity which consists of two elements in terms of software: a generated piece of code that has to be integrated with a service platform (*monitoring client*) and the monitoring event receiver being a part of the Virtual Service Platform (*monitoring server*) (see Figure 8).

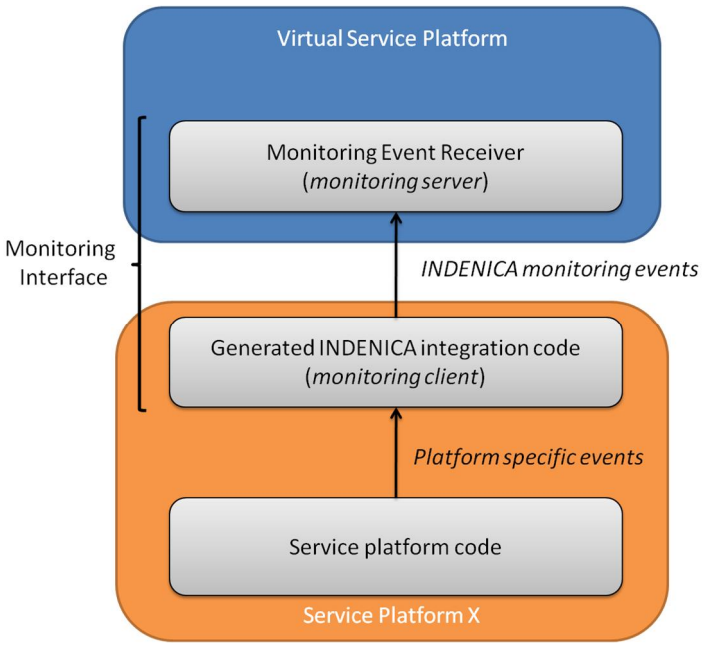

Figure 8: Monitoring Interface overview

The *monitoring client* receives events from the service platform and translates them into the INDENICA event model, based on rules generated from design-time tools. It is also responsible for sending these events to the monitoring server in the virtual service platform. The code for the monitoring client is generated by INDENICA design time tools, but it will require some manual integration with the platform code which has to be done once per platform. One specific part of the monitoring client will be the generic resource monitoring framework SPASS-meter which offers a uniform view on the resource consumption of the individual service platform in terms of its running services, components and dynamic variability.

The *monitoring server* (Monitoring Event Receiver) is a server-type component being a part of the VSP, which is able to receive events from remote service platforms. It accepts only events compliant with the monitoring events model, and passes them to the Monitoring Engine component.

The communication between *monitoring components* will be based on a publish/subscribe middleware, for example JMS or more sophisticated implementations, to allow for the complete decoupling between data sources and analyzers.

The JBoss Rule Engine (Drools) is a business rule management system (BRMS). It supports the JSR[4] standard for its business rule engine and enterprise framework for the construction, maintenance, and enforcement of business policies in an organization, application, or service.

[4] http://jcp.org/en/jsr/detail?id=94

4 Adaptation of the Virtual Service Platform

4.1 Adaptation Overview

In this section we describe the architecture and functionality of INDENICA's adaptation capabilities. Virtual Service Platforms as envisioned by INDENICA enable applications and services to view complex service environments as a unified service delivery platform independent of the subsystems' specifics. These requirements demand for integrated systems management functionality and support of flexible adaptation mechanisms. The complexity of today's IT systems is steadily rising, and especially large-scale, heterogeneous systems, consisting of a large number of subsystems, are difficult to manage effectively. To make such complex systems manageable, a high degree of automation is necessary to support system administrators. This high degree of freedom given to the system should be controlled by high-level policies and goals. Hierarchical structures allow for divide-and-conquer adaptation strategies and support different levels of abstraction. Thus, domain experts can provide their expertise where it is needed and are not overwhelmed by the overall complexity of the system.

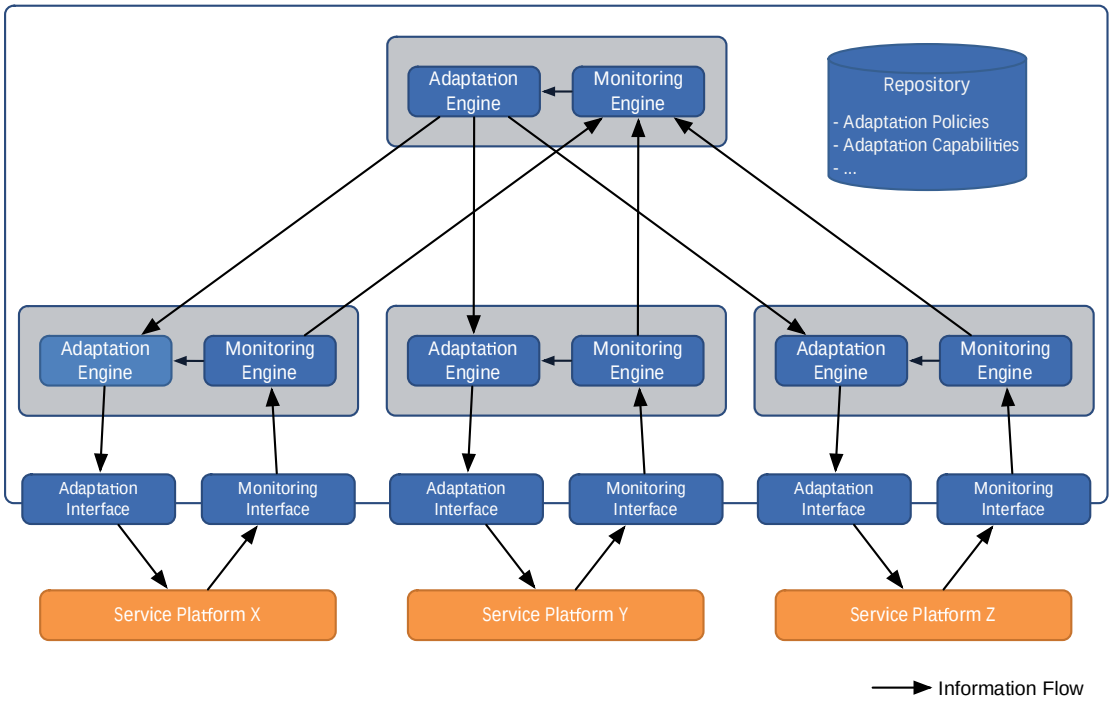

Figure 9: Adaptation Overview

Based on these requirements, we developed the INDENICA adaptation framework architecture shown in Figure 9. In order to support a wide range of service platforms, the INDENICA platform provides a generic adaptation interface for their integration. Platform providers use the adaptation interface to specify the adaptation capabilities of their respective platforms. Similarly, the monitoring interface is used to describe monitoring capabilities, which is used by the Monitoring Engine (cf. Section 3). The Adaptation Engine is responsible for executing adaptation policies, based on input

provided by the Monitoring Engine. There are different types of adaptation policies, e.g., specific rules, consisting of trigger and adaptation action, or high-level goals, which only express desired system states, without recipes stating how to achieve them. Adaptation policies and adaptation capabilities can be retrieved from the Repository, which is described in D2.3.1. The adaptation framework is designed in a layered manner, to allow for effective management of multiple service platforms. The architecture shown in Figure 9 represents an example configuration of the adaptation framework, and in general, monitoring and adaptation controller components can be arranged in arbitrary tree-like structures.

We will demonstrate the functionality of the adaptation framework using a simplified INDENICA use case, shown in Figure 10.

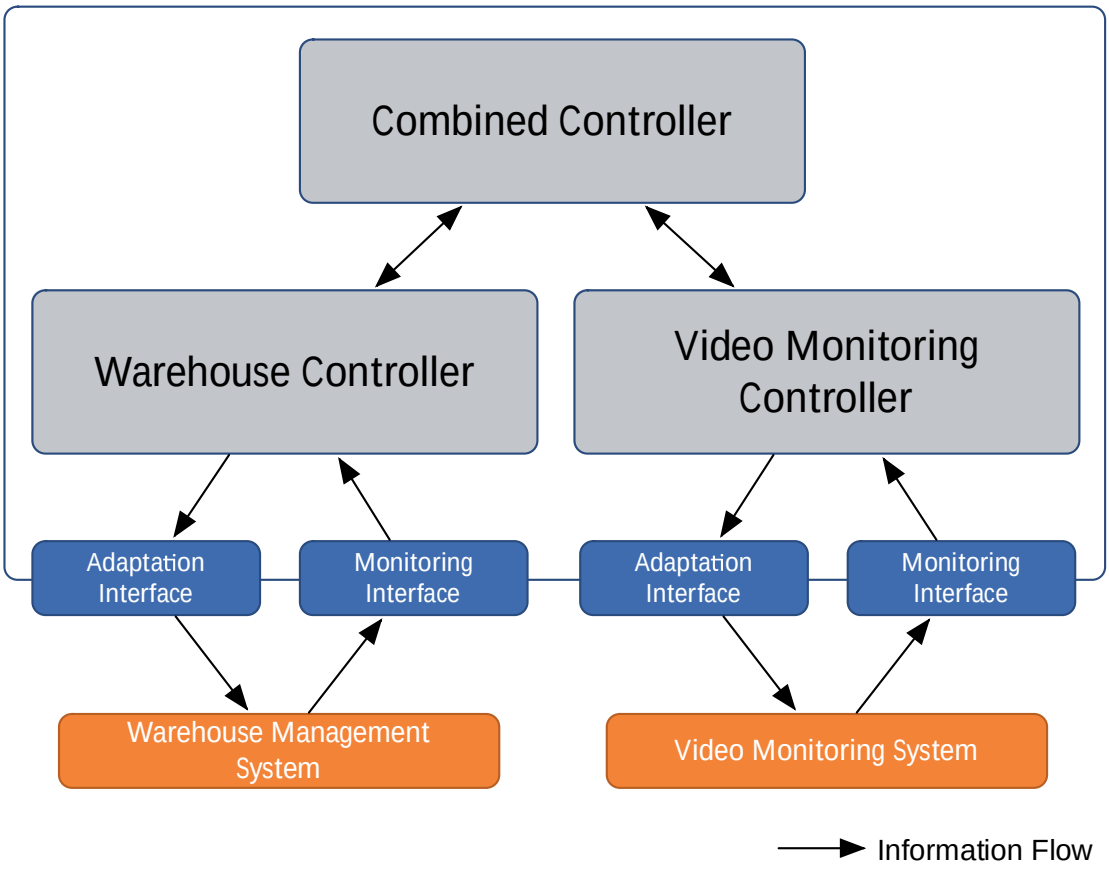

Figure 10: Adaptation applied to simplified warehouse use case

The scenario consists of an automated warehouse supporting different storage and retrieval strategies, which can be adapted. Additionally, a video monitoring system is used for surveillance and is able to identify incoming and leaving trucks. The Warehouse Controller is responsible for local management of the warehouse, such as reactions to failures in the transport system or lift modules. The Video Monitoring Controller is likewise responsible for local management of the Video Monitoring System. Moreover, it is responsible for adaptation processes, such as, video stream quality adjustments; therefore, influencing the detection rate of the truck identification mechanism. Both controllers process incoming monitoring data, and forward combined, analysed and filtered data to the combined controller. This allows the Combined Controller to make decisions from a more abstract point-of-

view, spanning multiple systems, exceeding a single controller's capabilities. As an example, suppose that the Video Monitoring Controller detects and forwards, that a high number of loaded trucks is about to arrive. The Combined Controller can then trigger an adaptation of the storage strategy in the warehouse using the Warehouse Controller; therefore, resulting in modification of the storage density of stored items.

4.2 Adaptation Model

The INDENICA adaptation model (AM) is designed as a set of layered Adaptation Engines, each implementing autonomic MAPE [8] managers. This facilitates separation of concerns, allowing for low-level Adaptation Engines to deal with granular changes in system behaviour, and high-level Adaptation Engines to focus on the specification of overall service level goals.

Traditionally, adaptation frameworks rely on predefined management policies, which are to be carried out. However, in complex distributed systems, management policies of different components can be conflicting, causing undesirable system behaviour and errors [9].

The INDENICA adaptation model allows for different levels of adaptation actions, as shown in Figure 11, represented as an escalation model ordered by invasiveness and degree of automation:

- Adaptation of concrete service platforms via the Adaptation Interface;
- Adaptation of the deployed VSP, using a capability model stored in the Repository, specifying possible adaptation actions for the deployed components;
- Adaptation of runtime variability as specified in the variability model;
- Adaptation of the models used to create the VSP instance;
- Notification of an Administrator.

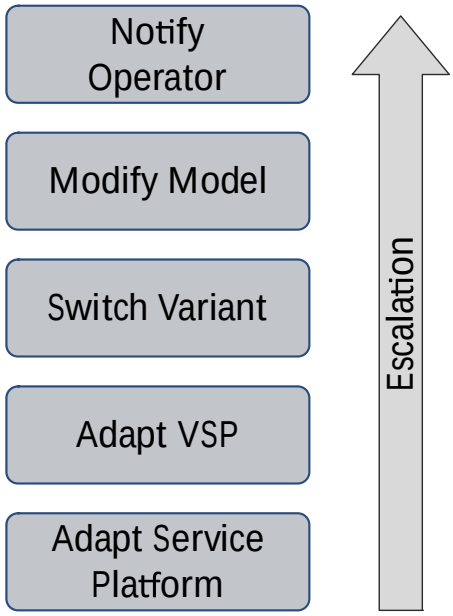

Figure 11: Levels of Adaptation

In order to provide for dependability and reliability, the model allows for the definition of escalation scenarios for adaptation actions. The adaptation framework will always try to fulfil high-level adaptation goals by executing the most effective and the least invasive actions; moreover, if no suitable automated measures can be taken then it will notify the system administrator.

Adaptation actions performed by administrators can also be monitored. Furthermore, if it is possible, adaptation actions can extend the used policies to gradually increase the self-healing capabilities of the VSP.

4.3 Adaptation Engine

An Adaptation Engine (AE) is responsible for carrying out adaptation actions. Its basic architecture is shown in Figure 12.

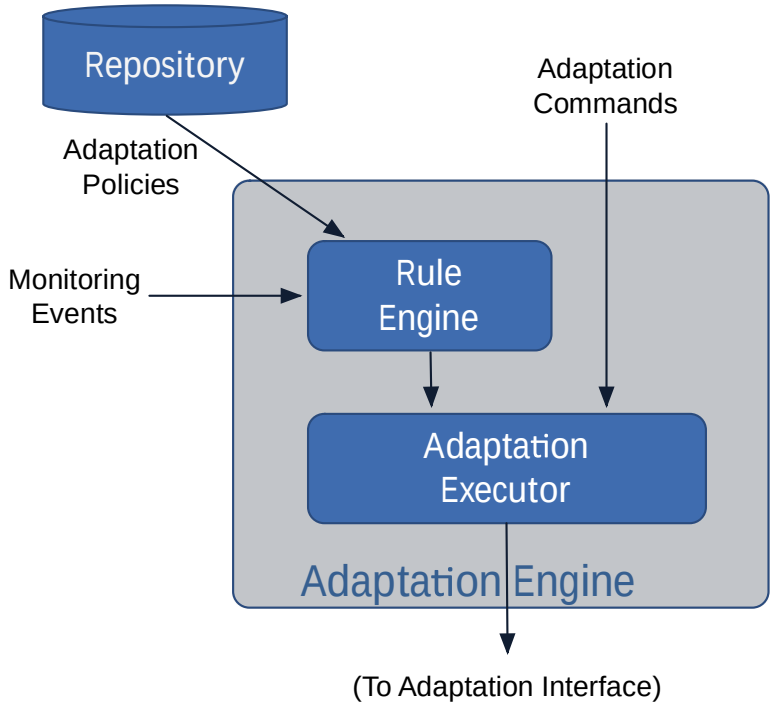

Figure 12: Adaptation Engine Overview

The AE contains an exchangeable rule engine and an execution component, providing an adaptation interface for higher-level AEs. Furthermore, the AE is designed in an extensible manner, allowing for the introduction of new functionality. Rule-based adaptation policies specify monitoring events, relevant for deciding if adaptation actions are necessary. An AE subscribes to all relevant events, and the rule engine is invoked whenever new events arrive, to evaluate the rule's conditions. When a policy rule matches, the adaptation executor carries out the adaptation action by invoking the adaptation interface, directly executing the action in the target platform or lower-level AE. As mentioned before, this layered approach allows developers and system administrators to specify abstract service level goals in high-level AEs, which need not be concerned with specifics on how to actually achieve these goals. Low-level AEs, on the other hand, can be created with a focus on specific platforms, incorporating expert knowledge about individual systems, providing optimized adaptation actions for fulfilling high-level goals.

To evaluate the capabilities of the proposed approach, we extended a high-level AE by adding a Policy Optimization component, able to detect and prevent policy conflicts and system failures, shown in Figure 13. This is achieved by deriving a Markov Decision Process (MDP) representation from the gathered monitoring data and observed policy actions (log data). The Policy Creator is then able to employ machine learning techniques to optimize the created MDP representation, thus generating an optimized management policy.

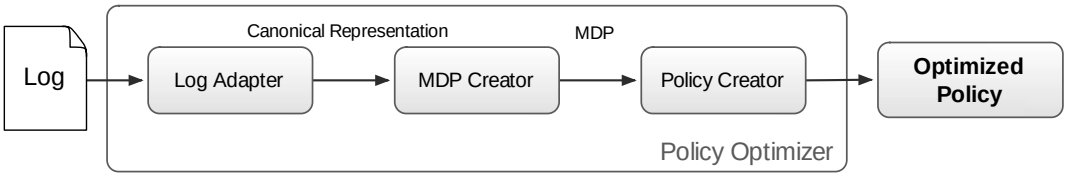

Figure 13: Policy rule optimization component

4.4 Adaptation Interface

The INDENICA platform provides a generic adaptation interface (AI) for service platforms to be integrated. Platform providers use the AI to specify the adaptation capabilities of their respective platforms, and to map INDENICA adaptation commands to platform-specific actions.

The AI is based on a capability model, allowing for the description of all available adaptation actions, annotated with information about their cost, failure probability, preconditions, and effects. Platform providers can store all available information about adaptation actions using the capability model.

As illustrated in Figure 14 the Adaptation Interface consists of two elements: the adaptation interface SCA component which is a part of the INDENICA platform, and an INDENICA adaptation commands translator which is a piece of code located on the target service platform.

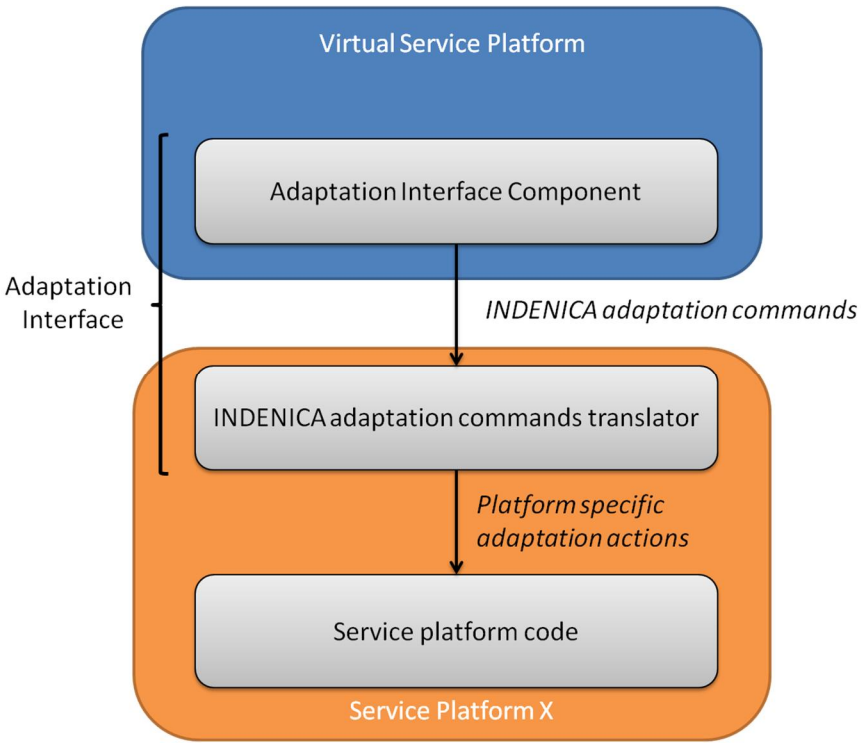

Figure 14: Adaptation Interface overview

The AI SCA component provides a unified interface for passing adaptation commands to integrated service platforms. Integrated service platforms are identified either by ID or type, the adaptation actions to be performed are specified in the capability model, and necessary parameters can be supplied. The adaptation commands are then passed to the INDENICA adaptation commands translator running on the target service platform.

The INDENICA adaptation commands translator is a piece of code which receives adaptation actions sent by the AI SCA component and maps them onto platform specific adaptation actions. The communication part of this code is rather generic and could be generated by INDENICA tools; however the actual integration of this piece of code with the service platform requires manual modifications of the service platform. These modifications are required to:

a) Bundle the generated code with the service platform (e.g. so that it is started and stopped together with the service platform).
b) Translate received INDENICA adaptation commands into service platform specific adaptation commands, which also include adding actual code which is able to execute those adaptation actions.

The procedure for integrating the Adaptation Interface with a service platform is a step that will need to be done only once per platform, to make it INDENICA compliant.

4.5 Prototype Implementation

We have implemented a prototype of the architecture presented in this section, as shown in Figure 15.

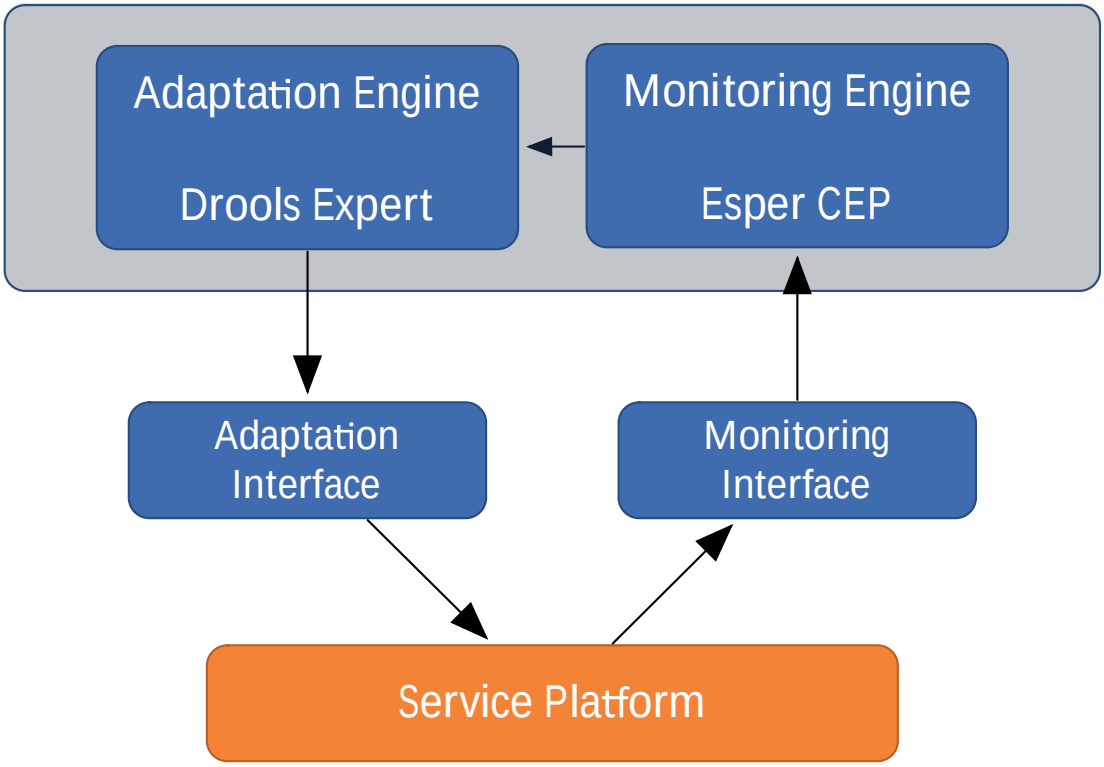

Figure 15: Prototype implementation architecture

The adaptation and monitoring modules are realized as SCA components, based on the Apache Tuscany framework. The Monitoring Engine utilizes the Esper[5] Complex Event Processing Framework, which offers a Domain Specific Language for event processing, the Event Processing Language (EPL), which allows for dealing with high frequency time-based event data. The Adaptation Engine employs the Drools Expert[6] rule engine, a business rule management system with a forward chaining inference based rule engine, i.e., a production rule system using the Rete algorithm.

The runtime configuration of the prototype is shown in Figure 16, describing the components that constitute the system. Furthermore, references between components, as well as service endpoints are defined in the configuration.

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

[6] http://www.jboss.org/drools/drools-expert.html

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      targetNamespace="http://WP4Runtime"
      xmlns:hw="http://WP4Runtime"
    name="WP4Runtime">

    <component name="ComponentInitializerComponent">
        <implementation.java
            class="wp4.deployment.ComponentInitializerImpl" />
        <reference name="monitoringEngine"
            target="MonitoringEngineComponent" />
        <reference name="monitoringInterface"
            target="MonitoringInterfaceComponent" />
        <reference name="adaptationEngine"
            target="AdaptationEngineComponent" />
        <reference name="adaptationInterface"
            target="AdaptationInterfaceComponent" />
        <reference name="repository" target="RepositoryComponent" />
    </component>

    <component name="MonitoringEngineComponent">
        <implementation.java
            class="wp4.monitoring.component.MonitoringEngineImpl" />
        <reference name="adaptationEngine"
            target="AdaptationEngineComponent" />
        <reference name="repository" target="RepositoryComponent" />
    </component>

    <component name="MonitoringInterfaceComponent">
        <implementation.java
class="wp4.monitoringInterface.component.MonitoringInterfaceImpl" />
        <reference name="monitoringEngine"
            target="MonitoringEngineComponent" />
        <reference name="repository" target="RepositoryComponent" />
    </component>

    <component name="SamplePlatformMonitoringInterfaceComponent">
        <implementation.java
class="wp4.monitoringInterface.component.SamplePlatformMonitoringInte
rfaceImpl" />
        <reference name="monitoringEngine"
            target="MonitoringEngineComponent" />
        <reference name="repository" target="RepositoryComponent" />
    </component>

    <component name="AdaptationEngineComponent">
        <implementation.java
            class="wp4.adaptation.component.AdaptationEngineImpl" />
        <reference name="adaptationInterface"
            target="AdaptationInterfaceComponent" />
        <reference name="repository" target="RepositoryComponent" />
    </component>

    <component name="AdaptationInterfaceComponent">
        <implementation.java
class="wp4.adaptationInterface.component.AdaptationInterfaceImpl" />
    </component>

      <component name="RepositoryComponent">
        <implementation.java
```

```
            class="wp4.repository.component.RepositoryImpl" />
        <reference name="adaptationEngine"
            target="AdaptationEngineComponent" />
    </component>

      <component name="RandomNumberGeneratorPlatformComponent">
            <implementation.java
      class="wp4.sampleplatform.RandomNumberGeneratorPlatformImpl" />
            <reference name="monitoringInterface"
            target="SamplePlatformMonitoringInterfaceComponent" />
      </component>
</composite>
```

Figure 16: Prototype runtime configuration

Figure 17 shows an exemplary query used by the Monitoring Engine, calculating the availability of the 'ERPService' during the last 24 hours. The Monitoring Engine uses this query to create a `ServiceAvailabilityReportEvent` containing the value of the calculated `avail` variable in the `actualAvailability` attribute.

```
select
      1 - (
      (select count(invocations)
      from ServiceInvocationFailedEvent(service='ERPService')
            .win:time(60*60*24.0) as invocations )
      /
      (select count(invocations)
      from ServiceInvocationEvent(service='ERPService')
            .win:time(60*60*24.0) as invocations ))
as avail
```

Figure 17: Sample EPL Query calculating service availability

The Adaptation Engine uses the adaptation rule shown in Figure 18 to evaluate the `ServiceAvailabilityReportEvent` generated by the Monitoring Engine, to determine, if the current system state requires corrective measures. In this particular example, the Adaptation Engine attempts to increase the service platform's redundancy level if the measured availability drops below 99.99%. The adaptation action `increaseRedundancyLevel` is interpreted by the adaptation interface, which in turn performs the actual corrective measures on the service platform.

```
rule "When availability too low, increase service redundancy level"
when
  $e : ServiceAvailabilityReportEvent( actualAvailability < 0.9999 )
then
  NotificationActions.notifyOperator($e);
  AdaptationActions.increaseRedundancyLevel();
end
```

Figure 18: Sample DRL adaptation rule

5 Deployment of Virtual Service Platforms

5.1 Design-time requirements and Platform Variability

This chapter provides an overview of the deployment manager and its requirements, its functionality and the deployment process itself. The deployment manager is in charge of packaging, publishing and starting the service platforms as well as the virtual service platform itself. To fulfill these tasks, different kind of information is required.

Platform providers have to describe the respective deployment process of their platform in terms of a generic deployment script because no limitations on used technology for the service platforms should be superimposed. The deployment-specific variability as described in the generic deployment script will be resolved at deployment time and the instantiated deployment scripts will be used for packaging, publishing and running the service platforms.

Information from the service component view, the service deployment view and the runtime view (respectively the more specific artifacts generated by the generation tools) are used together with adapters and descriptors for the monitoring framework to package the Virtual Service Platform. Finally, deployment scripts for publishing and starting the packaged virtual service platform have to be generated and executed.

5.2 Deployment Process

This section describes the process of the deployment manager (DM). The domain of the DM is to package the virtual service platform and deploy it to the runtime environment. The process is started by the platform integrator in the course of deploying the virtual service platform or in the course of a *RegenerateAndRedeploy* adaptation activity (cf. D3.1, section 3.3).

The prerequisite for the deployment process is that the variability of the individual service platforms has to be resolved until deployment binding time. The runtime variability will be left, because no assumptions should be superimposed about the service platform target environment. The following process is divided into two main stages, namely instantiation, packaging and deployment of 1) of the individual service platforms to be integrated and 2) the virtual service platform.

As a first activity, the monitoring glue code will be generated. For the creation of artefacts for the monitoring framework, the platform provider has to supply mappings between platform-specific events and events used by the INDENICA platform (cf. section 3.4). This information is used by the DM to generate glue code as far as possible to adapt the monitoring- and Adaptation Engine of every platform contained in the virtual service platform (Figure 14). To unify the monitoring view on the resource consumption of every platform, particularly of the activated variants realized as services or components, the generic resource monitoring framework SPASS-meter is used. The DM generates a monitoring scope definition for SPASS-meter based on the monitoring requirements, the variability model and the asset model. Furthermore, the DM triggers the static instrumentation process of SPASS-

meter to insert monitoring probes according to the scope definition into the respective platform. The next step is to package and publish the service platforms. Thereby, deployment time variability will have already been bound using information from the variability model (WP2), the artefact model (WP2) and the service deployment view model (WP3). Generic deployment scripts, supplied by the platform provider, are instantiated by tools from WP2 and WP3 and used by DM for the concrete packaging and publishing of the service platforms together with the monitoring artefacts.

The second stage concerns the instantiation and deployment of the virtual service platform. This stage is divided into three main activities, namely 1) the packaging of the virtual service platform including the monitoring and adaption framework, 2) the generation of deployment descriptors for the virtual service platform and 3) the final publishing of this platform. First, all artifacts of the Virtual Service Platform are packaged by the DM into a deployable format called SCA Contribution as defined as [10]. This includes all SCA descriptors, the virtual service components, as well as the artifacts concerning the monitoring and adaption framework. The Service Deployment View, respectively the generated UML 2 deployment model, contains vital information about the deployment process itself and will be interpreted by the DM. Based on this information, the deployment-ready packages are processed. Deployment scripts are generated that are tailored to the existing infrastructure (as described in the Service Deployment View). After everything is generated and packaged, the actual deployment process of the virtual service platform begins. The service platform and the technology platform will be deployed at once using the generated deployment scripts, which contain all steps necessary to publish and start artifacts at specific nodes, according to the deployment diagram. By executing these scripts, the deployment process is finalized. For executing this activity, Maven is used, but the process is not specific to Maven and could be extended to be used with other deployment tools as well. The Maven script contains information about the target environment, the artifacts that need to be deployed and additional dependencies. More specifically, it contains a mapping of which artifact needs to go to which runtime environment.

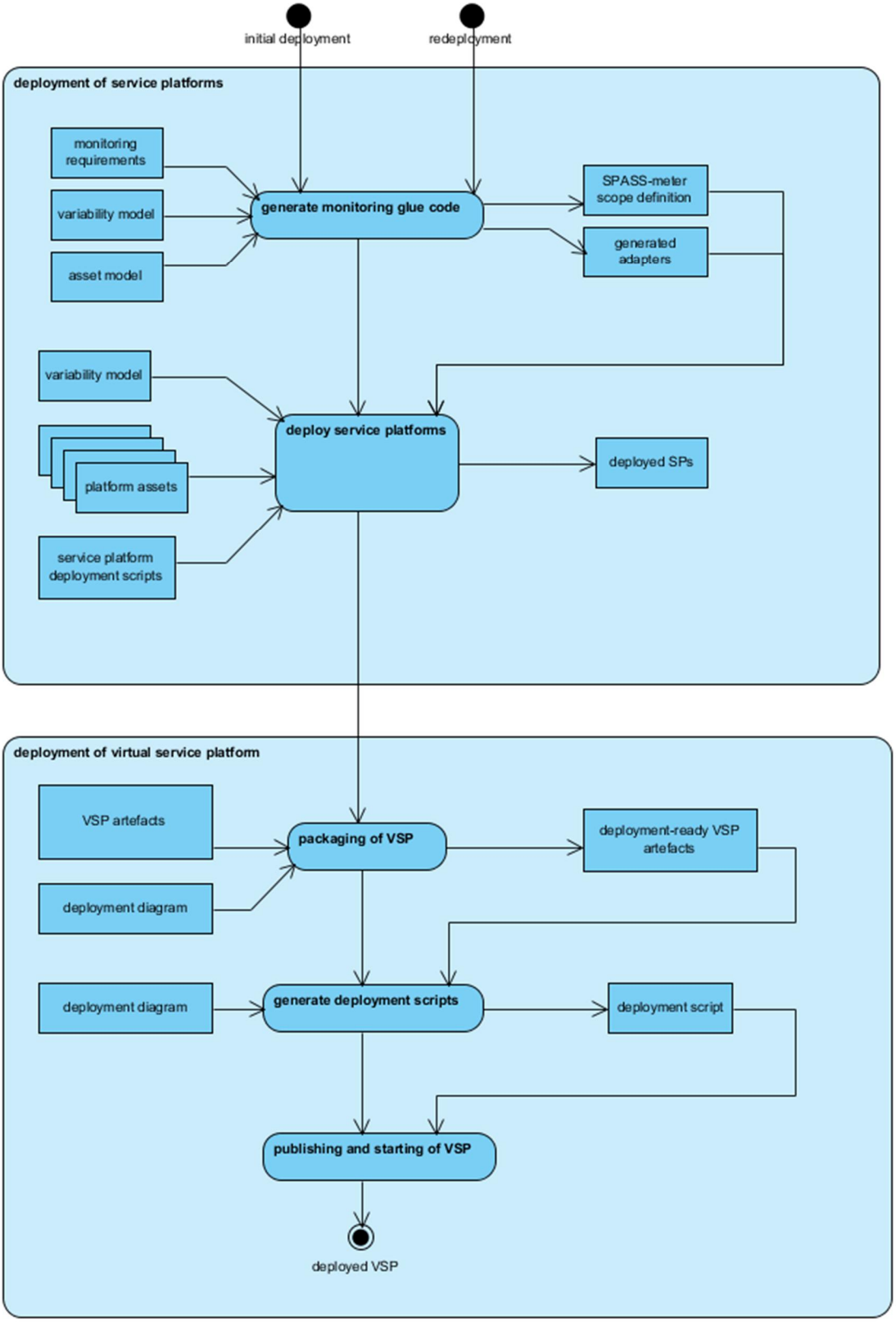

Figure 14: Deployment Process

5.3 *Deployment descriptors*

The Deployment Descriptors for the Virtual Service Platform as well as the individual service platforms describe how the (partially) configured and instantiated platforms

will be prepared for packaging, how they are packaged, where they will be published and how they have to be configured to be used.
In our case we can identify three main groups of Deployment Descriptors. The first group describes the single service platforms, the second the SPASS-meter instance responsible for monitoring the platforms and the last group the deployment to the virtual service platform.
The Deployment Descriptors for the single service platforms are platform specific and supplied by the platform provider. They comprise a variability description and will be instantiated with the Variability Resolver.
The descriptors for the generic resource monitoring using SPASS-meter contain the general configuration of the instrumentation process as well as the monitoring scope. The monitoring scope lists the individual services and components as well as the concrete resources to be monitored. This information can be derived from the monitoring specification from the requirements model, from the variability model, the related asset model and via the asset model also from the architecture models. Even if the information in this descriptor will primarily be used at deployment time, the information will be part of the deployed instance in case that (additional) runtime instrumentation might be needed, e.g. to consider services and components to be added dynamically at runtime.
The descriptors for deploying SCA components, modeled in the Service Component View, to the runtime of the Virtual Service Platform are an output of the Generation (see D3.1). Also these descriptors can be divided into three groups. First for description of the physical structure of the SCA domain mainly derived from the information of the Service Deployment View. The description comprises the information about all nodes of the domain, their binding and the assignment of SCA composites to the nodes. The second group contains descriptors holding the information about the logical structure of the SCA domain which was modeled in the Service Component View. One descriptor comprises all SCA composites of the SCA domain whereby each SCA composite has a descriptor assemble its SCA components. The third and last group contains SCA metadata at one hand for the SCA domain at the other hand for each SCA contribution. This information is derived from Service Component View and Service Deployment View.

6 Integration with Other Work Packages

6.1 Work Package 1 - Requirements Engineering & Methodology for Interoperable Service Platforms

Needless to say, the requirements elicitation activities carried out in WP1 are responsible for specifying the requirements of the "integrated" platform-to-be, and thus also for the actual Virtual Service Platform. These requirements address functional and non-functional properties, but also the needs for adaptation and variability. All of them will be used to design the platform. The more formal/precise these requirements are, the more fruitfully they will be used in the next phases.

First of all, these requirements will serve to test the platform and understand whether the implementation satisfies stated requirements. Even if the project does not comprise any specific activity about testing, this is clearly a very important and interesting phase.

Since the requirements elicitation is not limited to the functionality of the platform and its qualities of service, the information provided by WP1 will also be used as guidance for engineering the variability of the platform-to-be and also for the actual specification of the monitoring and adaptation artefacts.

The idea here, depending on the formality of provided specification/information, is that variability annotations will be used to design the actual variability embedded in the design of the solution, and also the one that must be dealt with at runtime. Also monitoring directives and adaptation plans are then derived directly and used to feed the INDENICA infrastructure. Monitoring directives are in the form of assertions and more generally statements that must hold true on the system, adaptation plans come in the form of steps the system must undertake to keep itself on track.

6.2 Work Package 2 - Variability Engineering

Variabilities represent options or alternatives in software development assets such as requirement documents, design models, source code or even executables. The derivation of a concrete product (here a service platform) from variable assets happens by binding the variabilities, e.g. defining which alternative should be enabled. Binding of variability may happen at various points in time during the software development lifecycle. These so called binding times specify the latest point in time when a variability may be bound. Furthermore, the binding of a variability may be constrained by dependencies among variabilities. Dependencies may restrict the binding of individual variabilities or prescribe the concrete binding dependent on previously bound variabilities. Variabilities will be derived from the requirements model defined in WP1 and are described using the variability modelling tool in WP2 in terms of a variability model and an associated asset model which links variabilities to affected assets.

A subset of the variabilities in INDENICA platforms will be bound at design time (WP3) by selecting appropriate architectural styles, at generation time of the virtual platform or the connectors to the technical platforms (WP3, WP4), at compilation

time of the platforms, or at deployment time. The remaining variabilities to be bound later than deployment time will be left open to be resolved at runtime (as a partially instantiated variability model). A subset of the runtime variabilities can be used to describe the adaptivity of a platform (Dynamic Software Product Line approach).

The instance of a VSP is a composition of the configured technical platforms based on their individual variability models (WP2, WP5). The remaining variabilities which are not instantiated during the development or deployment of the VSP will be input to WP4 as a partially instantiated variability model. Unbound runtime variabilities and the dependencies among them define the adaptation space and, thus, the possible decisions of the adaptation manager. Changing the concrete binding of runtime variabilities in a technology platform will enact the decision of the adaptation manager via the technology-platform-specific monitoring and controlling adapter. The variability model and the asset model may be used to configure monitoring activities, particularly those regarding quality and resource properties in the technical platforms using the generic resource monitoring framework SPASS-meter (see also Section 0).

WP2 will provide the specification of the variability and the asset model for the implementation and integration activities in WP4 (intended are also APIs for directly accessing the partially instantiated WP2 models from the WP3 Repository). Parts of the monitoring activities (e.g. those carried out by SPASS-meter) may be automatically configured based on the information from WP2. As an overall constraint, the decisions of the Adaptivity Engine (WP4) must be consistent with the dependencies and constraints in the variability models of the individual technology platforms (from WP5).

6.3 Work Package 3 - View-Based Architecture and Tools for Tailoring Service Platforms

The major contributions of WP3 are a view-based design time and runtime architecture and its tooling that support stakeholders in dealing with complexity and heterogeneity of service platforms through the notion of virtual service platforms (VSP). The view-based architecture aims at providing the stakeholders different view models for representing VSPs from different perspectives and abstraction levels at design time and runtime. These view models are derived from the common, abstract concepts of a Core model, and therefore, are able to be linked to each other via the Core model. Apart from that, code generation techniques and templates are also developed for producing code, configurations, and/or runtime monitoring and adaptation directives (see D3.1 [5]).

The components presented in the previous sections such as Monitoring Engine, Adaptation Engine, and/or Deployment Manager shall naturally reference to the concepts and elements, especially the Runtime View, provided by the view-based architecture (as described in D3.1 [5]). For instance, which service components of VSP are going to be monitored? Which QoS measurements are applied for such service components? Which service components are going to be changed or reconfigured to adapt to a certain new situation?

Moreover, runtime monitoring directives and queries as well as adaptation rules can be (semi-)automatically produced based on view models (and/or their extensions) and code generation techniques provided in the view-based architecture Described in D3.1 [5].

7 Conclusions and future steps

In this document we present the complete view of the framework, which will act as an integration layer for all the technical work on concrete models and tools done in INDENICA project. It will also be used in the case study as a solid foundation for the proof-of-concept realisation. The presented framework description depicts architecture for the realisation of the WP4 tool suite and describes the relation between these tools.

Although the document is in line with the completion of Milestone 2, which means that the interim concept of the framework is established, some work on early prototyping has been already done. This document will help the consortium in further development of the framework by providing a generally accepted view of the WP4 platform on the technical and conceptual level. However, the consortium takes into account that some minor changes in the concept might occur due to the fact that the technical work goes beyond current State of the Art and in some cases novel concepts will need to be included.

Throughout the development of the framework, significant focus will be put on the scalability of the proposed solution and awareness of the possible future exploitation of the framework (i.e. integration with additional underlying service platforms). Another key aspect of non-functional requirements for the framework itself will be to provide pre-requisites for every segment of the framework in which additional integration will be possible in the future (i.e. exchanging or adding new components).

With regards to the agreed Description of Work, during the next 6 months the main focus of work in Work Package 4 will be concentrated on two tasks: 1) the development process of the tool suite for the framework, including the initial integration with the tools and concepts from remaining Work Packages; 2) finalization of the Framework concept.

Table of Figures

References

[1] Haleh Mahbod, Raymond Feng and Simon Laws. „Java Feature — What is SCA? A quick view of concepts through and an example walkthrough“. Java Developer Journal, February 4, 2007. http://soa.sys-con.com/node/325183

[2] Haleh Mahbod, Raymond Feng and Simon Laws. „Building SOA with Tuscany SCA. A simple service-oriented infrastructure,“. Java Developer Journal, November 9, 2007. http://java.sys-con.com/node/458183

[3] Yehia Taher, Marie-Christine Fauvet, Marlon Dumas, and Djamal Benslimane. 2008. Using CEP technology to adapt messages exchanged by web services. In Proceeding of the 17th international conference on World Wide Web (WWW '08). ACM, New York, NY, USA, 1231-1232. DOI=10.1145/1367497.1367741 http://doi.acm.org/10.1145/1367497.1367741

[4] Supreet Oberoi. „Introduction to Complex Event Processing & Data Streams“. SOA World Magazine, October 1, 2007. http://soa.sys-con.com/node/434463

[5] INDENICA Deliverable D3.1. *View-based Design Time and Runtime Architecture for Tailoring VSPs,* 2011-09-30.

[6] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. 2008. Advanced event processing and notifications in service runtime environments. In Proceedings of the second international conference on Distributed event-based systems (DEBS '08). ACM, New York, NY, USA, 115-125. http://doi.acm.org/10.1145/1385989.1386004

[7] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. 2010. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. IEEE Trans. Serv. Comput. 3, 3 (July 2010), 193-205. http://dx.doi.org/10.1109/TSC.2010.20

[8] Tesauro, Gerald, "Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies" *IEEE Internet Computing*, 11(1):22 – 30, Jan./Feb. 2007.

[9] E. Lupu and M. Sloman. *Conflicts in policy-based distributed systems management*. IEEE Transactions on Software Engineering, 25(6):852 – 869, Nov/Dec 1999.

[10] SCA Service Component Architecture, Assembly Model Specification, SCA Version 1.00, OSOA Collaboration, 15th March 2007, http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications