Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 1 (2010) 125–133
Procedia Computer Science
www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Dynamic monitoring framework for the SOA execution environment

Daniel Żmuda[a,*], Marek Psiuk[a], Krzysztof Zieliński[a]

[a]Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland

Abstract

The paper analyses the challenges involved in constructing a dynamic monitoring framework compliant with the requirements of SOA application monitoring. The specification of these requirements provides a starting point for our multilayer framework architecture. We describe its Monitoring Scenario and Instrumentation layers in detail. The approach aims at runtime monitoring of container-based SOA execution environments. The Instrumentation layer exploits interceptor sockets, on top of which a powerful attribute-based reconfigurable monitoring service is constructed. The proposed solution is application-agnostic and can be used for enterprise and computational applications. We present a case study which further explains the functionality of the system. A prototype implementation for OSGi containers is also briefly described. Preliminary performance evaluation results are outlined and discussed.

Keywords: Monitoring tools, SOA monitoring, dynamic run-time reconfiguration, interceptor pattern

1. Introduction

Service Oriented Architectures (SOA) are about improving the accuracy of IT solutions in terms of fulfilling business requirements [1]. A crucial aspect of this accuracy is responding to constant changes in business requirements at runtime. SOA is perfectly capable of handling such needs, as service composition can be dynamically rearranged while individual services do not have to be shut down. Such dynamism introduces requirements for SOA management. The management layer and related monitoring should be able to refocus to the new elements which become important following composition rearrangement. There is a need for a solution which can dynamically choose the subject and the scope of monitoring at runtime and provide the management layer with the appropriate monitoring data. The goal of this study is responding to the presented need. This is done through the following means: (i) designing a generic architecture capable of dynamic, reconfigurable runtime monitoring; (ii) implementing a prototype of this architecture; (iii) evaluating the prototype in the context of functional and non-functional aspects.

The presented study adopts an event-based approach and Complex Event Processing [2], which is important element of Event-driven SOA [3]. The designed framework is based on the adapted interceptor pattern, where the interceptors are exposed as services. The interceptor, when accompanied by additional features, can be plugged into

*Corresponding author
Email addresses: daniel.zmuda@agh.edu.pl (Daniel Żmuda), marek.psiuk@agh.edu.pl (Marek Psiuk), kz@ics.agh.edu.pl (Krzysztof Zieliński)

1877-0509 © 2010 Published by Elsevier Ltd.
doi:10.1016/j.procs.2010.04.015

	R1	R2	R3	R4	R5	R6	R7
1. W. Branimir et al. [5]		+			*?*	*?*	
2. C. Momm et al. [6]		+		*?*	*?*	*?*	
3. O. Tiejun et al. [7]		+		*?*	*?*	*?*	
4. X. Bai et al. [8]	+	+	+				
5. Swordfish [9]	+			+		+	+

Table 1: Comparison of related research with the formulated requirements

the monitored environment to provide monitoring data on an on-demand basis. The set of plugged interceptors is determined by the *monitoring scenario*, which admits a declarative model-based definition of monitoring scope and its goals. The *monitoring scenario* is based on a model environment topology, attribute-based service discovery and metrics. What is important, the *monitoring scenario* can be dynamically changed. It is assumed that the container provides fundamental service-oriented features (for example *service discoverability*) and that the container can be instrumented for the purposes of monitoring. The Enterprise Service Bus (ESB) is very often used as a middleware platform for SOA development and there is trend to apply a Open Services Gateway initiative (OSGi) container [4] as a basis for the implementation of new, innovative ESB products. It has therefore been decided to realize the prototype presented in this paper for OSGi-based ESB container.

The paper is organized as follows: Section 2 formulates the requirements and describes recent related work. Section 3 describes the technology-agnostic architecture of the framework. Subsequently, in Section 4, an evaluation of the architecture on the basis of the implemented prototype is provided. The paper ends with conclusions.

2. Requirements and related work

To perform a complete analysis of related work, requirements for the presented approach are discussed. This enables comparison of the related work with the requirements (cf. Table 1) in order to clearly describe the value added by the solution proposed in this paper. The formulated requirements are an implication of the framework's goal and are enriched, especially in the non-functional scope, with focus on performance issues. The functional and non-functional requirements are as follows:
Functional requirements:

- (**R1**) On-demand, runtime monitoring of arbitrary services exposed in the execution environment;
- (**R2**) Declarative model-based specification of the monitoring goal, changeable at runtime;
- (**R3**) Architecture which can be extended in the direction of both system and business monitoring.

Non-functional requirements:

- (**R4**) Only necessary data needed to fulfill the monitoring goal is intercepted;
- (**R5**) Monitoring mechanisms attempt to not influence the performance of unmonitored elements and performance deterioration of monitored elements remains minimal;
- (**R6**) Instant responses to changes in the monitoring goal and environment topology;
- (**R7**) Instrumentation must be transparent and performed on demand.

The subject of monitoring and its dynamic aspects have been already widely addressed in pre-SOA systems. The studies [10, 11] present approaches of monitoring distributed Java applications, where monitoring request can be served on-demand thanks to specialized JVM agents or JVM library instrumentations. The study [12] presents challenges of grid monitoring, where dynamism and selectivity are important demands answered by proposed solution. The experience from mentioned studies is taken as basis for the research in dynamic SOA monitoring which is not yet comprehensively addressed. However there is ongoing research which separately addresses issues related to the subject of dynamic SOA monitoring.

W. Branimir et al. [5] propose a generic approach for monitoring BPEL-based service compositions. They introduce a domain-specific language for expressing the process performance model (PPM), which is then transformed and fed into the Process Engine and the Monitoring Tool. Monitoring data is acquired by means of events generated by the Process Engine. To implement it, the PPM has to be mapped to the event model of a particular engine and a

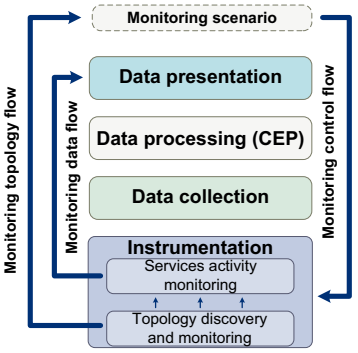

Figure 1: Layered system architecture

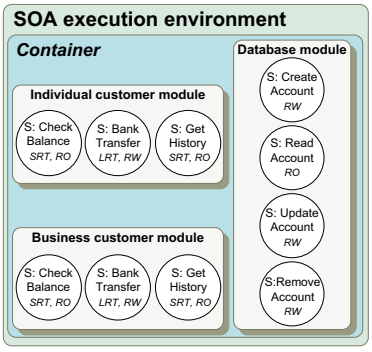

Figure 2: Topology of a sample banking application

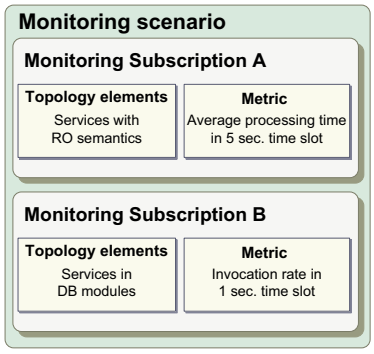

Figure 3: Sample monitoring scenario structure

deployment descriptor of each process has to be provided along with lists of events required for monitoring. Enriching deployment descriptors implies that a change in a model requires changing the descriptor and redeploying the process. Such lack of dynamism is similar to approaches presented in [6, 7], which are also leveraging the events generated by a process engine. This is different from the study presented in this paper, which focuses on the capability of dynamically changing the monitoring scope and subject at runtime. Nevertheless, all three papers exploit some model (a model of virtual business process or PPM), which is common with the approach presented in this paper.

The work of X. Bai et al. [8] deals with SOA dynamism in the context of testing. Their goal is to propose a SOA testing solution which can automatically adapt to any dynamic reconfiguration of service bindings (monitored environment) and also to the reconfiguration of test scenarios. Although the work of X. Bai et al. is not related to monitoring, it remains similar to the approach presented in this paper. Both approaches use events for the purposes of triggering reconfiguration. Moreover, the subscription/notification pattern is similar. However, there are also significant differences. The study of X. Bai performs testing by means of agents, while this study uses the instrumentation and interceptor pattern for monitoring purposes. The approach of X. Bai et al. can follow the reconfiguration of service binding but changes in service inventory (adding, removing services) are not covered in the discussed article. Owing to attribute-based discovery of the monitoring subject, the approach proposed in this paper adapts to any changes in service inventory at runtime.

Table 1 confronts formulated requirements with the presented related work. The + symbol means that the requirement is satisfied, the *?* symbol means that the subject is not covered and the blank field means that requirement is not satisfied. As depicted, none of the presented developments fulfills all stated requirements. The approach presented in this study (similarly to the Swordfish) uses the interceptor pattern but aims at fulfillment of all the formulated requirements.

3. Framework Architecture

This section describes the layered architecture of the presented framework. It also introduces specific architectural patterns such as model-based declarative monitoring goal specification called *monitoring scenario* and an instrumentation mechanism named *interceptor socket*. The presented architecture is generic - it is not bound to any specific technology.

3.1. Architecture concept

The architecture of the designed framework is depicted in Figure 1. The *monitoring scenario* represents an abstraction of a monitoring goal and drives the entire monitoring process. There can be one or more scenario and each can be separately activated or deactivated. Below the *monitoring scenario* are four architectural layers. Some elements of the architecture exchange essential monitoring information, referred to as the *monitoring flow*. The architecture uses the term topology, which, in the scope of this paper, means a collection of services, modules and containers present

in the monitored SOA environment. It is assumed that modules are deployed to containers and services are registered by modules. Therefore, complete information about a topology includes knowledge about relations between topology elements, i.e. to which container a given module is deployed and by which module a given service is registered.

The functionality of each architecture layer is as follows:

- instrumentation layer - it consists of two sublayers: i) topology discovery and monitoring , ii) service activity monitoring. The former is responsible for discovering the container topology state, while the latter uses this state to effect monitoring of particular services.
- data collection layer - it defines mechanisms of monitoring data distribution and gathering. In general, this layer acts as a backbone for collecting data from distributed containers.
- data processing layer - it uses CEP to process monitoring data and calculate metrics. In CEP, a processing task is defined by a statement. Therefore, creating and installing appropriate statements enables aggregation and correlation of elementary events provided by the data collection layer.
- data presentation layer - presents the activity of services and the topology state to the end user.

The introduced *monitoring flows* have the following meaning. *Monitoring topology flow* indicates that the instrumentation layer provides the *monitoring scenario* with information about topology state. This state could be important for defining the monitoring goal. *Monitoring control flow* represents the monitoring process driven by the *monitoring scenario*. The set of active *monitoring scenarios* determines which services are monitored. Individual *monitoring scenarios* can be activated or deactivated at runtime, which triggers dynamic reconfiguration of the instrumentation layer. Finally, the *monitoring data flow* indicates that data acquired from monitoring particular service activities is transmitted by the data collection layer, processed by CEP components and presented to the end user.

Due to space limitations, this paper does not describe all elements of the architecture in detail. It focuses on the concept of *monitoring scenarios* and dynamic instrumentation layer reconfiguration mechanisms named *interceptor sockets*. They are described in the following sections.

3.2. Monitoring scenario model

The model of a *monitoring scenario* introduces the following elements: *monitoring subscription*, *metric definition*, *metric attribute*, *metric instance* and *target topology specification*. The *monitoring scenario* encompasses one or more *monitoring subscriptions* and specifies the scenario time frame, i.e. when the scenario is active. The *metric definition* defines the metric name and indicates which *metric attributes* have to be provided when the metric is instantiated. The actual values of *metric attributes* represent the metric configuration, which, for example, can specify how the metric monitoring data should be aggregated. The *metric definition*, accompanied by particular values of *metric attributes*, forms a *metric instance* which can be expressed by a CEP statement. The *monitoring subscription* encapsulates a *metric instance* and a *target topology specification*. The *target topology specification* can be provided in two ways:

- explicit - exact definition of topology elements by providing particular element identifiers, i.e. service names or service modules,
- implicit - indirect definition of topology elements by providing a set of element attributes.

Implicit specification is realized on the basis of the model of attribute-based topology element characteristics. The model assumes that each topology element can be described via attributes. There is a set of attribute types and each type can have one or more values. Values can be assigned to a topology element and the set of assigned values forms the element characteristic. Attributes of deployed services are assigned whenever services are designated and, following deployment, are stored in a *service attribute registry*. Thus, it is possible to match them with those formulated in specific attribute-based model instances.

To explain how the *monitoring scenario* model operates, a topology of a sample banking application (cf. Figure 2), along with an example of a *monitoring scenario* (cf. Figure 3), is presented. Each service of the banking application has a set of attributes. The *SRT* attribute stands for *short running transaction* and means that a service realizes a transaction whose duration is relatively short. The *LRT* attribute stands for *long running transaction*. The *RO* and *RW* attributes stand for *read-only* and *read-write* semantics respectively. In the sample *monitoring scenario*, the first *monitoring subscription* declares interest in services which provide short running transactions. This is an example of an implicit specification of topology elements. The second monitoring subscription specifies interest in all services

exposed in the database module of a sample application and therefore uses an explicit specification of topology elements by the module name.

Monitoring subscriptions depicted in Figure 3 encapsulate two *metric instances*. The first *metric instance* comes from the *metric definition* called processing time. The attributes, i.e. time slot and aggregation method (evaluating the *average*), are accordingly specified. The second *metric instance* of the invocation has only one *metric attribute*, namely the time slot. Activation of a defined *monitoring scenario* should result in the following monitoring activities for the sample banking application:

- the processing time of services *Check Balance*, *Get History* and *Read Account* is measured in accordance with a specified *metric instance* because services have the *read-only* property,
- the invocation rate of database module services is measured in accordance with a specified *metric instance* because the module is explicitly defined in the scenario.

The *target topology specification* can also be provided partially implicitly and partially explicitly. For example, the specification could point out all services with the *SRT* property, which are registered by the database module. The explicit topology specification implies that the subscription remains valid as long as the specified topology element is present in the container. On the contrary, the implicit specification is durable, i.e. it does not depend on the presence of topology elements. Therefore, if a new service is registered in the container and its attributes match the *topology target specification*, it will be dynamically covered by the monitoring process.

3.3. Interceptor socket mechanism

The *interceptor socket* mechanism is realized by means of container instrumentation, which, contrary to application or service instrumentation, is transparent from the point of view of deployed modules. The proposed design assures that the mechanism can be used to achieve any monitoring level of the SOA execution environment. The *interceptor socket* mechanism is similar to the solution present in the Swordfish framework; however, there are significant differences which are highlighted in the following paragraphs.

The *interceptor socket* relies on the Aspect Oriented Programming (AOP) instrumentation of the service exposition process. When the service is exposed, the instrumentation injects the code of the *interceptor socket* and wraps the exposed service with a *socket*. Any service invocations goes through the socket and can be intercepted if needed. The implementation of such a mechanism is not trivial and has to be adjusted for particular containers; however it is always realizable because the principles of service-orientation [1] ensure that the process of service exposition always takes place.

The *interceptor socket* consists of two components: service observer and interceptor chain. These two components are instantiated upon the first invocation of a service. The key purpose of the socket is to enable plugging-in of interceptors. An interceptor is a system component which has interest in topology elements, has a priority and processes service invocations in accordance with the implemented processing logic. The main assumption is that the interceptor is implemented as a service and publishes its interest and priority. It allows the service observer of the *socket* to be notified by container-specific discoverability mechanisms about interceptor registration and unregistration. When the observer is notified about a new interceptor, it evaluates if the interceptor is interested in the service wrapped by the socket. If the interceptor's attributes match socket attributes, that interceptor is added to the *interceptor chain* of the analyzed socket in accordance with the interceptor's priority. Similarly, when the interceptor is unregistered, the observer is notified and the interceptor is removed from the *interceptor chain*. The socket provides information about every intercepted service invocation by the sequential socket's interceptor chain evaluation.

The proposed monitoring solution specifies two different types of interceptors: i) *agnostic interceptor* - as the name implies, this type of interceptor focuses mainly on service invocation rather that specific invocation parameters, so that the overhead generated by agnostic interceptors is minimal, ii) *business interceptor* - unlike the agnostic interceptor, it is interested in particular service invocation content or parameters and processes this data in accordance with the implemented business logic. In this paper we focus on agnostic interceptors while the use of business interceptors is part of our future work, heading towards higher layers of the SOA execution environment.

Figure 4 presents the interceptor chain modification process following activation of the *monitoring scenario* depicted in Figure 3. Framework mechanisms create and register new interceptors for specific subscriptions. The whole process is identical for both subscriptions, however in order to simplify the figure, only one interceptor is shown (Interceptor RO - the interceptor interested in services with *read-only* semantics). The new interceptor is registered as

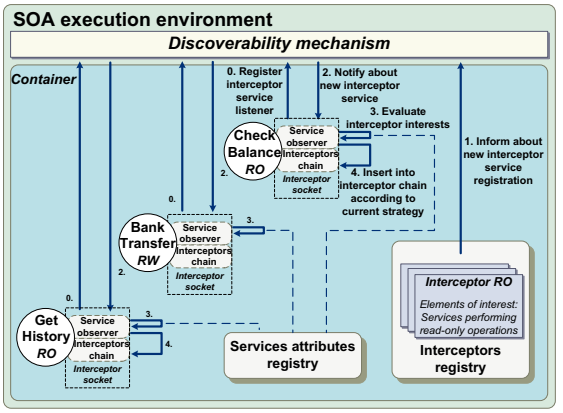

Figure 4: Assembling a newly registered interceptor service into sockets

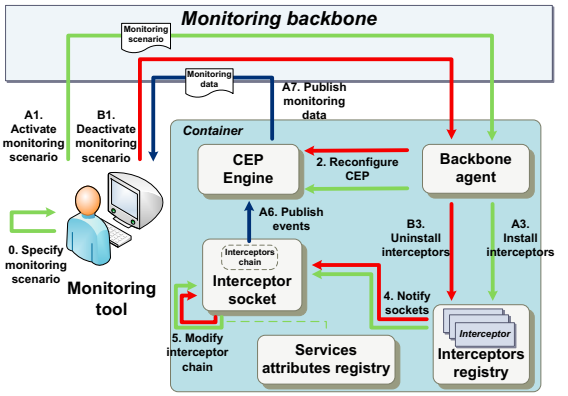

Figure 5: Monitoring scenario activation and deactivation logic

a service and the service observer of each *interceptor socket* is appropriately notified. Each socket evaluates whether the new interceptor is interested (either explicitly or implicitly) in the socket's service. In the case of implicit interest, the observer has to look up the *service attribute registry*. If the interceptor is interested, it is added to the socket chain; otherwise the chain is not modified. In the presented case, the sample RO interceptor is added to the sockets which wrap the *read-only* services (cf. Figure 2 and 3).

The Swordfish Framework [9] also uses the interceptor pattern but in a significantly different way. In Swordfish the interceptor chain is created each time a service is invoked. It is then processed and finally disposed of. The solution presented in this study moves the activities of chain creation and modification closer to the moment of interceptor installation. Such an approach reduces the overhead for monitoring services (the chain does not have to be created each time) and there is no overhead for services interceptors are not interested in (where the socket chain is empty). In the Swordfish, there is not any declarative way (similar to *Monitoring scenario*) of specifying what and how should be intercepted. Swordfish interceptors generate more overhead and are not as flexible as mechanisms presented in this study.

Combining the model of *monitoring scenario* and the *interceptor socket* mechanism is useful when approaching the problems of dynamic, transparent monitoring of an SOA environment. Each monitoring subscription of a scenario refers to a particular interceptor instance, which is configured for specified topology elements of the SOA execution environment and publishes data in accordance with a defined metric instance. Figure 5 illustrates the framework following the process of scenario activation (green arrows - A tag) and deactivation (red arrows - B tag).

4. Evaluation

Prepared evaluation use case is constructed for business domain, although presented solution is designed in the domain agnostic way, so it can be easily adopted for any domain, especially computational science. As it was mentioned earlier, more and more architectures for computational science are moving towards SOA paradigm, therefore presented framework answers the demand of monitoring of computational services.

The prototype is realized for an OSGi-based ESB container, namely ServiceMix 4.1. For the purposes of Complex Event Processing, the prototype uses the Esper Engine. OSGi introduces the model of dynamic service-oriented modularization of the code executed in a single JVM (Java Virtual Machine) process. In OSGi, code is deployed to the container by means of bundles. Bundles can expose and consume services thanks to the dynamic, flexible service model present in OSGi.

Due the dynamic nature of the service model provided by OSGi, the patterns used in the presented architecture were implemented with ease. The OSGi bundle is mapped to the *module* topology element from the previous section and, naturally, the OSGi service is a *service* topology element. The process of obtaining a reference to an OSGi service from the registry is instrumented by means of AOP on the basis of the preliminary work by M. Psiuk [13]. The

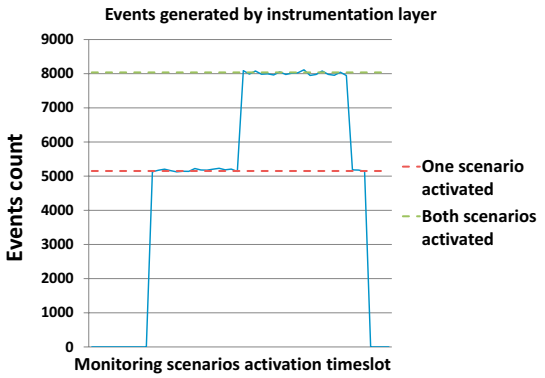

Figure 6: Event rate associated with activation and deactivation of monitoring scenarios

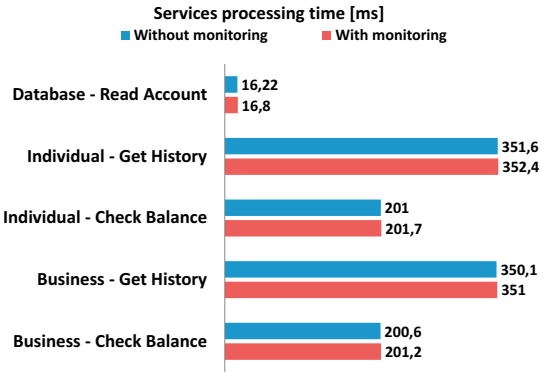

Figure 7: Overhead of monitoring process

reference returned by the OSGi registry is wrapped with the *interceptor socket*. The instrumentation assumptions about the structure of the container do not extend beyond the OSGi specification, which makes the instrumentation valid for any implementation of OSGi. All interceptors implement the agreed-upon interface and are registered as services with properties, which express interest in *interceptor sockets*. The interceptor service observer of an *interceptor socket* is realized by means of an OSGi service tracker, which listens for the interceptor interface. The observer is created upon the first invocation intercepted by the *interceptor socket*. The prototype is restricted to monitoring a single container but its design enables straightforward evolution towards distributed containers.

In the scope of the prototype, only agnostic interceptors have been implemented, along with the following related metrics: *invocation rate*, *error rate*, *processing time*. All metrics are measured at regular intervals (the *time slot*). Additionally, the *processing time* can be aggregated with the following well-known functions: *min*, *max*, *avg*, *std*, which is realized by means of CEP layer. The prototype implementation also includes the presentation layer, which is developed on the basis of Eclipse framework plugins. To enable communication between the presentation layer and the monitored container, the skeleton of data collection layer (backbone) has been implemented; however, the backbone, which uses the Java Messaging Service (JMS), cannot yet support the distributed ESB scenario. The UI provides three main views: *topology monitoring*, *monitoring scenario definition* and *results visualization*.

To evaluate functional and non-functional aspects of implemented prototype the banking application depicted in Figure 2 is used. Application modules are implemented as bundles, which expose respective services. The application is used simultaneously by ten individual customers and twenty business customers. The services belonging to the *database* module are invoked most frequently as they are used by higher-level services of *individual* and *business* customer modules. As the bank has more *business* customers, the services from the *business* module are invoked more frequently than those from the *individual* module. The application's database is optimized for reading; therefore *database* read service has the shortest processing time.

The case study considers the following possible situations that can occur in the application:
A) The Get History service is removed from the *business* module for maintenance purposes,
B) The Get Extended History service is added to the *business* module,
C) Some error occurs on the database and all services with the *RW* attribute throw exceptions.

On the monitoring framework side, the case study is set up as follows:
1) The user launches the UI interface for the monitoring framework and connects to the container.
2) The user opens the topology view to discover the environment topology. The bundles and services of the banking application, along with their state and attributes, are identified.
3) The user uses the *monitoring scenario definition* view and creates a monitoring scenario with the configuration depicted in Figure 3. The monitoring scenario is activated.
4) The user performs analysis of results with the use of the *results visualization*. All services related to the defined *scenario* are monitored. The user can observe that the services of the *business* customer are invoked more frequently

than those of the *individual* customer, while the *database* services are the busiest. It is also evident that the processing time of the *database* read service is the shortest.
5) Situation A) occurs in the application and the user can observe (in the *topology* view) that the Get History service is removed. Moreover, in *results visualization*, the invocation rate drops down to zero and the service is marked as removed.
6) Situation B) occurs in the application and the user observes the addition of Get Enhanced History and, in *results visualization*, a new plot appears. The plot starts with zero invocations and slowly rises over time.
7) Situation C) occurs in the application and the user observes that the processing time of the RW *database* services drops to zero. To gather more detailed information, the user creates a new *monitoring scenario* in which he wishes to monitor the *error rate* of RW services.
8) Once finished, the user deactivates both monitoring scenarios. The presented case study provides insight into a possible way of using the framework to monitor a dynamic SOA environment. The framework can follow changes in the environment as well as changes in the set of activated *monitoring scenarios* (req. **R1**, **R2**). During the study, service performance deterioration was also measured. When the service was monitored, there was some very low performance deterioration, which is depicted in Figure 7. As can be seen, the deterioration is minimal and does not depend on the service processing time (req. **R5**, **R6**). The number of generated events was also measured. As depicted in the Figure 6, the number of events increases when new *monitoring scenarios* are activated and decreases upon their deactivation. This proves that plugging interceptors into *interceptor sockets* on demand works properly (req. **R7**) and the monitoring is truly selective (req. **R4**). Introduced two types of interceptors: *agnostic* and *business* ensure that framework is expendable in direction of both system and business monitoring (req. **R3**). The evaluation has proven that the implemented architectural prototype can be successfully used to monitor a dynamic SOA environment and that it complies with all requirements.

5. Conclusions

Execution environments for SOA applications require monitoring and management mechanisms fully compliant with the dynamic nature of their activities. Dynamic software remains a very challenging issue, despite the progress made by such technologies as OSGi. The presented Dynamic Monitoring Framework attempts to creatively exploit the most powerful dynamic patterns, such as attribute-based dynamic discovery of services, late binding, event-based notification and dynamic loading of modules. All these fundamental mechanisms, used togeher with a well-know interceptor pattern implementation, result in a runtime-reconfigurable, lightweight and flexible monitoring framework, which can be implemented in the OSGi environment with ease.

The presented work creates a foundation for further development. The proposed single-container solution will be extended to a distributed implementation of the proposed Dynamic Monitoring Framework, necessary to fully address SOA application monitoring requirements.

Acknowledgment

The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

References

[1] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
[2] D. C. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[3] T. Yongzhong, L. Baotai, Design high reliable monitor and control system using event-driven soa philosophy, in: IT in Medicine & Education, 2009. ITIME '09. IEEE International Symposium on, Vol. 1, 2009, pp. 146–153.
[4] T. O. Alliance, About the OSGi Service Platform (July 2004).
[5] B. Wetzstein, S. Strauch, F. Leymann, Measuring performance metrics of ws-bpel service compositions, in: ICNS '09: Proceedings of the 2009 Fifth International Conference on Networking and Services, 2009.
[6] C. Momm, M. Gebhart, S. Abeck, A model-driven approach for monitoring business performance in web service compositions, in: ICIW '09, 2009.
[7] T. Ou, W. Sun, C. Guo, J. Li, Visualized monitoring of virtual business process for soa, in: ICEBE '08: Proceedings of the 2008 IEEE International Conference on e-Business Engineering, IEEE Computer Society, 2008.

[8] X. Bai, D. Xu, G. Dai, W.-T. Tsai, Y. Chen, Dynamic reconfigurable testing of service-oriented architecture, in: Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, Vol. 1, 2007.
[9] D. W. Oliver Wolf, Eclipse Swordfish an Open Source SOA Runtime Framework for the Enterprise, SOPERA GmbH Whitepaper.
[10] M. Bubak, W. Funika, R. Wismller, P. Mtel, R. Orlowski, Monitoring of distributed java applications, Future Generation Computer Systems 19 (5) (2003) 651 – 663.
[11] C. Seragiotto, T. Fahringer, Performance analysis for distributed and parallel java programs with aksum, Vol. 2, 2005, pp. 1024 – 1031 Vol. 2. doi:10.1109/CCGRID.2005.1558673.
[12] B. Balis, M. Bubak, Monitoring infrastructure for grid scientific workflows, 2008, pp. 1 –10. doi:10.1109/WORKS.2008.4723959.
[13] M. Psiuk, AOP-Based Monitoring Instrumentation of JBI-Compliant ESB, Services, IEEE Congress on.