CMST 21(1) ??-?? (2015) DOI:10.12921/cmst.2015.21.01.001

Middleware for Managing QoS Adaptation of SOA Applications

Jacek Psiuk, Krzysztof Zielinski

AGH University of Science and Technology
Faculty of Computer Science, Electronics and Telecommunications
Department of Computer Science
E-mail: {jacek.psiuk, kz}@agh.edu.pl

Received: 29 May 2014; revised: 01 October 2014; accepted: 14 November 2014; published online: 18 January 2015

Abstract: The paper describes an improvement over our previous work: the concept of an Adaptive SOA Solution Stack (based on the IBM S3 model) which applies an AS3 element pattern to S3 layers where the need for adaptation arises. The presented improvement, called the adaptation strategy management process, represents a solution that enables an Adaptation Architect to model Quality of Service (QoS) adaptation in a declarative manner, automatically deploy it into a running system and then monitor its execution. Its main objective is to allow the Adaptation Architects to view the adaptation process on a higher level of abstraction and employ adaptivity mechanisms in working applications in an easy way. This is accomplished by incorporating the adaptive application metamodel developed in the DiVA EU project and adjusting it to the SOA context. This paper explains the challenges involved in adaptation strategy management and proposes extensions to the DiVA metamodel. Subsequently, it presents a method by which the Adaptive Manager (a component of the AS3 adaptation loop responsible for making decisions about adaptation) can execute adaptation strategies in accordance with the adaptation model. The presented approach is evaluated in a case study, creating an adaptation strategy and monitoring its impact on an application prototype.

Key words: adaptation management, adaptation strategy, SOA, adaptive manager

I. INTRODUCTION

The Service Oriented Architecture (SOA) paradigm represents an innovative approach to application integration, conveying numerous benefits [1]. An SOA application is composed of autonomous and loosely-coupled services communicating with one another and exposing well-defined interfaces. Applications developed in accordance with SOA principles can be executed in highly dynamic environments.

It is desirable to enrich applications with mechanisms allowing them to monitor their working environment [2] and react to perceived changes. An application which is the object of adaptation must expose elements which facilitate adaptation enforcement: sensors providing information on the current context state and effectors exposing the application's variability (adaptation decisions are applied by means of effectors). A *configuration* is a set of specific decisions that can be applied for each available effector. The adaptation process is enforced by *adaptation strategies*. An adaptation strategy is a description of actions which, when executed, improve the QoS of a running system, i.e. improve the system's fulfillment of selected non-functional requirements without influencing compliance of its functional requirements. This is done by evaluating possible configurations and applying the one which guarantees the best QoS given the current context state. The concept of *adaptation management* includes support for the following three aspects: 1) describing and publishing monitorability and variability of designated parts of the application, which will play an active role in the adaptation process, 2) creating and validating ready-to-execute adaptation strategies which make use of the exposed variability, 3) executing and analyzing the available adaptation

strategies. Applications supported by adaptation management middleware are referred to as *adaptive SOA applications*.

Designing large-scale enterprise applications which need to efficiently operate in a changing environment is not a trivial task.

Several challenges are associated with the adaptation management process [1, 3, 4]. The first problem we focused on is the **C1) complexity** of the managed application. The more points of variability adaptation processes need to account for, the harder they are to manage. Adaptation strategy management should handle a large number of possible configurations in a user-transparent manner. Furthermore, a robust method for **C2) validation** of adaptation strategies must be provided, which is even more crucial for complex applications. Additionally, we would like to ensure **C3) dynamic** adaptation strategy installation as well as **C4) automation** of its execution. This should be performed without disturbing availability of the running application. Modification of strategies and dynamic reinstallation of the running system should also be supported. Moreover, we need to focus on **C5) consistency**: adaptation strategy execution must ensure proper causality between the application configuration selected for the current circumstances and the application's QoS. Finally, the Adaptation Architect should be able to manage adaptation strategies with **C6) ease**: strategies should be created in a declarative way, hiding technical details from the Adaptation Architect. The problem of **C7) scalability** should also be addressed here: the adaptation strategy definition should retain its simplicity in more complicated scenarios, including many variability points.

The SOA Solution Stack (S3) [5] proposed by IBM is one of the possible approaches to modeling SOA systems. S3 provides a detailed architectural definition of an SOA system split into nine layers. The Adaptive S3 (AS3) model [3] has been constructed by uniformly applying the AS3 element pattern to each layer of the S3 Model. The Adaptive Manager is a component of the AS3 adaptation loop which makes decisions on configurations that should be applied to the running application in order to improve QoS in the current context state. As explained in [3], the Rule Engine is a good choice for Adaptive Manager implementation. However, the issue of creating rules which would execute a desirable strategy has not heretofore been addressed. This issue is considered in this paper.

This work includes the following scientific contributions:

- An adaptation strategy management methodology for QoS adaptation in SOA applications which addresses non-functional requirements compliance;
- A method for creating rules for the Adaptive Manager to enable it to execute the defined adaptation strategy;
- A middleware framework implementing the proposed solution, focusing on the Service layer of the S3 model.

The paper is organized as follows. Section II covers adaptation management aspects: a metamodel that has been adopted and extended to meet new requirements, an adaptation execution method by means of a Rule Engine, and a proposed methodology of adaptation enforcement. Section III presents a case study and evaluates the proposed solution. Work related to adaptation is recounted in Section IV. The paper concludes with a description of to-date achievements, outlined in Section V.

II. ADAPTATION MANAGEMENT

In this section we describe in detail the entire process of adaptation management: 1) what information needs to be gathered in order to prepare a specific adaptation strategy, 2) how to extract an executable strategy from the gathered information, 3) what steps the Adaptation Architect should take to create an adaptive application.

II. 1. Adaptation metamodel

In order to manage adaptation, a certain way of defining it has to be specified. On the one hand, it must allow the Adaptation Architect to easily create adaptation strategies; on the other hand, it has to be transformable to a form which can be executed by the Adaptive Manager. Substantial research has been done in the field of modeling variability and there are different approaches to achieving adaptivity. Some of them exploit "utility functions" as the main tool for specifying adaptation strategies, decision rules, model-driven and feature-driven approaches. We have decided to incorporate the solution developed in the EU DiVA project [6] because, unlike other examined approaches [4, 6, 7, 8, 9, 10], it substantially addresses challenges C1, C2, C5, C6. The adaptation metamodel[1] specifies elements of the *adaptation model*. The adaptation model is a description of elements participating in the process of adaptation: the context by which changes in the environment are captured, the variability (means of introducing changes into the application) and the causal link between the two (i.e. how to configure variability elements according to the context).

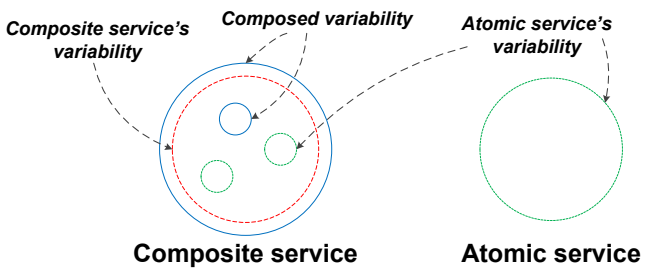

Fig. 1. Atomic and composite service variability

The DiVA adaptation metamodel needs to be extended in order to meet the requirements emerging in the context

[1] The original metamodel is briefly described in the Appendix: Section V.

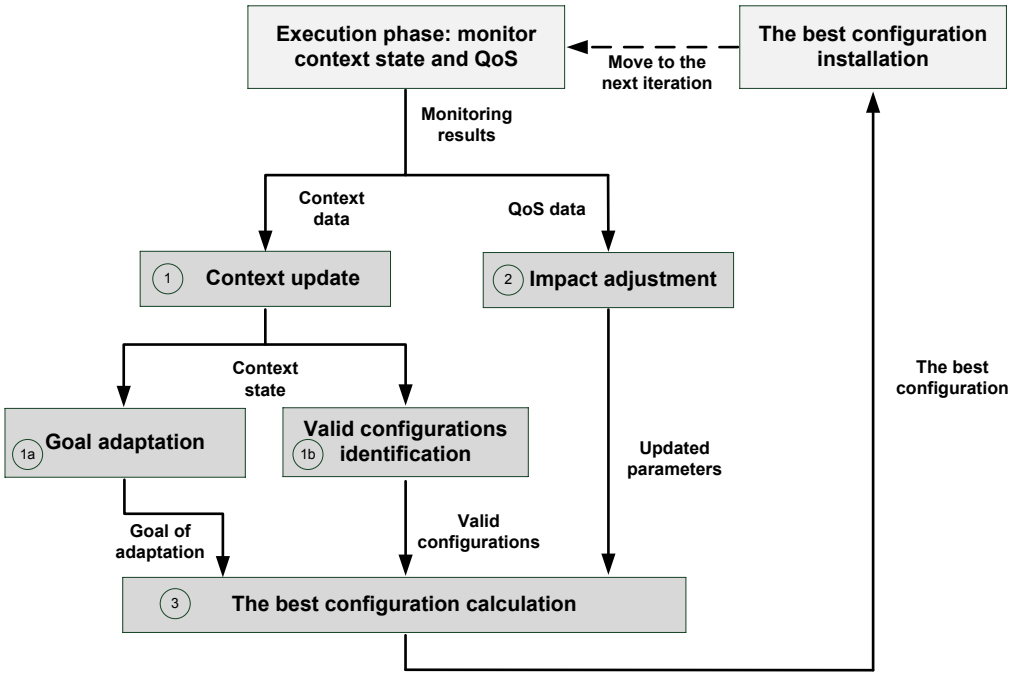

Fig. 2. Reasoning stages

of SOA. Originally the adaptation model was assumed to be created at design time and never change at runtime [11]. However, in light of challenge C3, highly dynamic environments must be taken into account, therefore it should be possible to alter the adaptation strategy at runtime. Secondly, we have upgraded the variability metamodel with regard to challenge C7. In most cases an SOA application consists of a number of services. In contrast to atomic services, composite services require cooperation with other services (whether atomic or composite), leading to a call graph. An SOA application's variability is a sum of the variability of all of its operating services. Variability of a composite service is influenced by its own variability and the variability of directly and indirectly referenced services [12]. Such a composite service should expose the variability altogether as composed variability (see Figure 1). This issue was not addressed in the DiVA metamodel since it was designed for Dynamic Software Product Lines [11]. Therefore the metamodel had to be extended with exposition of aggregated variability: adaptation parameters can enclose other adaptation parameters so that the Adaptation Architect can separate different areas of adaptation and group the overlaying ones. Finally, the goal metamodel needs to be revised in the context of SOA, where different QoS techniques exist [13]. We have therefore added the aspect of monitorability to the adaptation goals. The presented concept merges the context metamodel with the goal metamodel: context variables can be simultaneously treated as adaptation goals.

II. 2. Executing the adaptation

The reasoning process which results in new management decisions is performed by the Adaptive Manager. Therefore, with regard to challenge C4, the adaptation model has to be transformed into a specific adaptation strategy: a set of rules that are to be executed by a Rule Engine. During execution monitoring data from the context and QoS is gathered. Reasoning is triggered as soon as a change in the application environment occurs. The reasoning process is divided into steps (the algorithm is depicted in Figure 2) which are related to different rule sets:

1. The monitoring results are consumed by the rules and an updated context state is returned.
   - 1a. The goals' importance is determined according to the current context state.
   - 1b. Possible system configurations are evaluated and valid ones are identified.
2. According to the monitored values of QoS, the impact of installed variants is refined in order to better correspond to real influence on goals,
3. Given the current context state and a set of valid configurations together with the updated impact of the parameters of adaptation, the rules specify the best configuration.

Following the reasoning process the best configuration is installed in the running application and monitoring can continue.

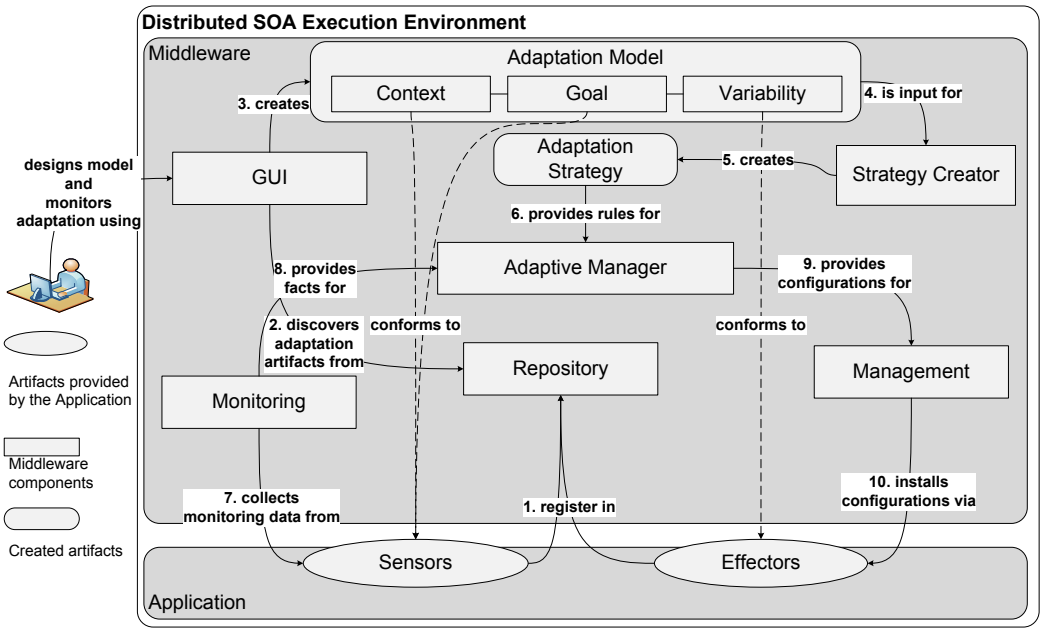

Fig. 3. Adaptation management process enforced by the middleware

II. 3. Methodology

This subsection presents the adaptation management process supported by the developed middleware whose architecture is depicted in Figure 3. Numbers in parentheses correspond to those marked in the figure showing the process. The middleware is deployed together with the application in an SOA environment which can be distributed over remote nodes.

Phase 1 – Adaptation artifacts deployment

The Adaptation Architect starts with designating which parts of the running application will actively participate in the adaptation process. Three adaptation artifact types are distinguished here: context sensors, QoS monitors and effectors. Context sensors and QoS monitors are types of sensors. The difference between them lies in their role in the adaptation model: context sensors become context variables, while QoS monitors are the goals of adaptation (unlike context sensors, their values are naturally ordered from the most to the least desired). The Adaptation Architect has to implement respective interfaces for every adaptation artifact. The implementation process is out of the scope of this paper (e.g. regarding sensors, methods presented in [13] can be exploited; regarding effectors, internal structure, configuration or collaboration with different services can provide different variants). Every active element must be registered in the Repository provided by the middleware framework (1).

Phase 2 – Adaptation modeling and simulation

The Adaptation Architect revises the adaptation requirements of the application and constructs an adaptation model that will fulfill these requirements during execution. Two GUI components were prepared for this phase: *Modeling View* where the Adaptation Architect can prepare adaptation strategies and *Simulation View* where he/she can simulate the prepared strategies. The Adaptation Architect discovers available sensors by means of GUI (2) and adds a subset related to this particular adaptation strategy to the model (these stand for context variables). QoS monitors as well as effectors are also discovered and added to the model (as goals and parameters, respectively). Following this step the Adaptation Architect completes the model by adding the remaining elements: goal management policies, constraints and impact.

Afterwards, the Adaptation Architect simulates adaptation to see what the results of running the adaptation according to that model would be. Prior to being transformed into an executable form, the model is validated by the middleware: its structural correctness is verified (information about any missing elements is displayed to the Adaptation Architect); simulation of every possible context state may expose defects (existence of context states for which none of the goal management policies apply; variants which are always (or never) chosen in the system configuration; context states for which there is no valid configuration to choose due to, for example, overly restrictive constraints). This completes creation of the model (3). It is then received by the Strategy Creator (4) which automatically derives an Adaptation Strategy (5) and installs it in the Adaptive Manager as rules (6).

Phase 3 – Execution

The Monitoring component collects data from sensors (7) and transforms it to facts (8). The Adaptive Manager chooses

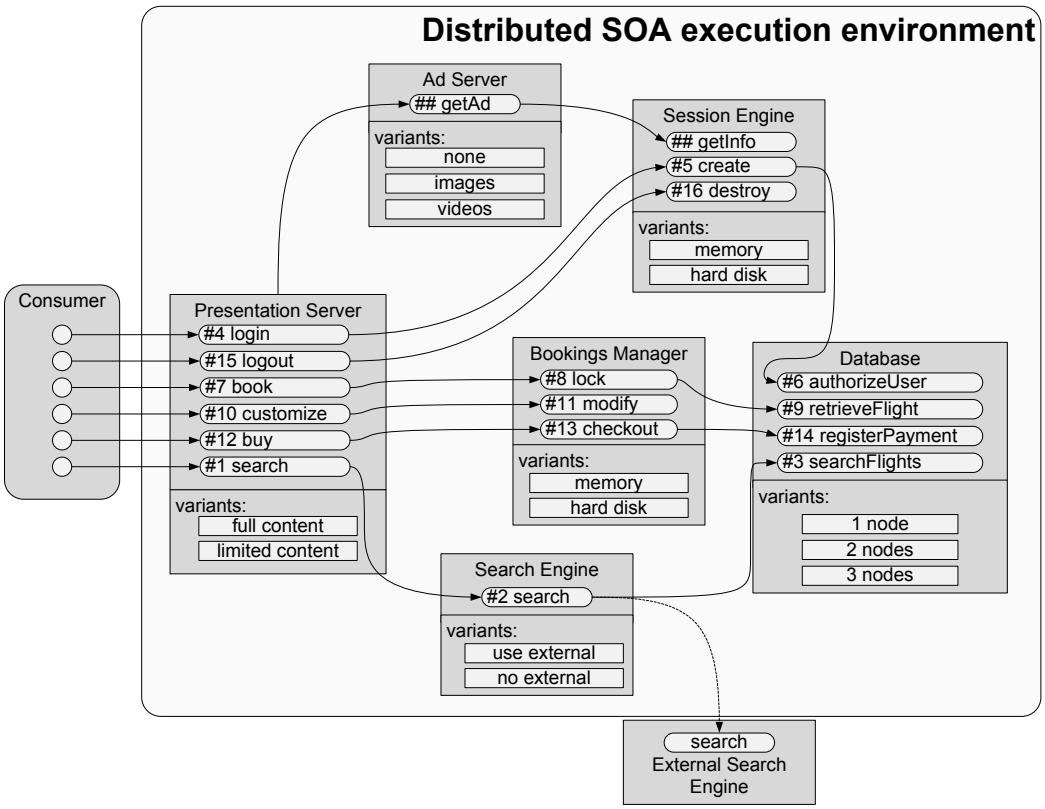

Fig. 4. Abstract view of the case study application showing the request flow and service variants

configurations accordingly (9) and they are installed in the Applications via the Management component (10). The Adaptation Architect can observe the execution progress via the *Execution View* where strategy execution results are shown (showing the context state, chosen configurations and QoS). If necessary, he/she can go back to Phase 2, modify the model and reinstall the strategy at runtime.

III. EVALUATION

Our work included the creation of middleware supporting the presented methodology[2]. This section presents an evaluation of the presented solution which demonstrates a prototype SOA application facing some issues related to performance and resources usage. We show how this particular case study can be enriched with an adaptation strategy that helps the application adapt to changing environment conditions in order to achieve better QoS. The testbed on which the approach was evaluated consisted of three OSGi containers: one with the middleware services; the application services were distributed over two remaining OSGi containers. The communication between separated OSGi containers was implemented using the mechanisms presented in our earlier work [14].

III. 1. Creating the adaptation strategy

The target application is designed to manage flight bookings, allowing clients to log in, search for available flights, book, customize and finally purchase tickets. The system is designed in accordance with the SOA principles and thus it consists of several services installed in an SOA environment and providing various features. Each service comes in several variants, differing with respect to QoS. It is assumed that only one variant can be chosen at any time for each service. Figure 4 shows an abstract view of the application. The request flow was highlighted to indicate typical service usage and cooperation. Business operations as well as service variants are also shown.

Consumers of the application send requests directly to the *Presentation Server* which later passes control to other services. As the name suggests, the Presentation Server is responsible for interpreting consumers' input and providing them with results. We assume that it can operate in two different variants: *full content* when the application serves requests providing its full feature set at the cost of higher resource

[2] Technologies used include OSGi-Equinox, Eclipse RCP and the JBoss Drools Engine.

usage, and *limited content* where resource usage is reduced. *Search Engine* is used to coordinate the process of searching for flights. It provides two variants: *use external*, using other searching services from outside of the application to achieve better offers, and *no external* when it uses only the local database to save time and bandwidth. Both services: *Session Engine* (managing client sessions) and *Booking Manager* (reserving flights from the moment they are looked up to the point of a booking request) may operate in one of two variants: *memory* and *hard disk*. These variants differ with respect to data storage mechanisms: memory storage (as opposed to hard disk) makes the service fast but consumes more operating memory which might be in short supply. The *Database* service provides persistence features. Since it may represent a performance bottleneck, it can be delegated to one, two or three nodes, depending on the average response time and the number of requests to handle (for the purposes of the prototype nodes were implemented as threads serving incoming requests). The last service, *Ad Server*, is responsible for serving advertisement to clients. It is invoked during every request to the Presentation Server. The Ad Server has three variants: *video*, *images* and *none*, providing various types of advertisements (or the lack thereof).

The adaptation requirements (ordered by importance) are: 1) to be able to adapt to the application's current response time and request count in order to maintain a sufficient level of responsiveness without wasting resources, 2) to maximize profit resulting from ticket sales and advertisements, 3) to release resources whenever they are not needed in order to minimize costs, including memory and bandwidth. Consequently, the application's *response time* and the *number of currently active clients* will be monitored by sensors, which are at the same time context sensors and QoS monitors. In addition, other sensors will provide context information: 1) *high memory usage* and 2) *high bandwidth usage* indicating if the memory or bandwidth usage is too high and should be reduced; and 3) *marketing time* indicating that advertisement profit is more important (for example, due to holiday discounts).

In order to achieve desirable adaptation, we follow the adaptation management process described in Subsection II. 3. In phase 1 we ensure that all sensors and effectors are properly deployed and registered in the Repository (1). In phase 2, using the Modeling View, we discover installed context sensors, QoS monitors and effectors (2) and add them to the model as context variables, goals of adaptation and adaptation parameters[3]. We complete the model with constraints, parameters' impact on goals and goal management policies[4]. Subsequently, using the Simulation View, we validate and simulate the adaptation and modify the model if necessary. Once the model meets adaptation requirements (3), the middleware converts it to an executable set of rules (4, 5) and installs it in the running system (6). In phase 3, we can follow adaptation execution using the Execution View (7-10). After a new strategy is installed, we can go back to phase 2, readjust and reinstall the strategy without stopping the running application.

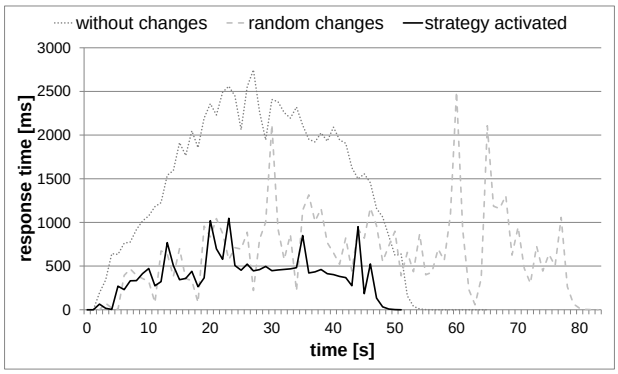

Fig. 5. Response time measured during the experiments

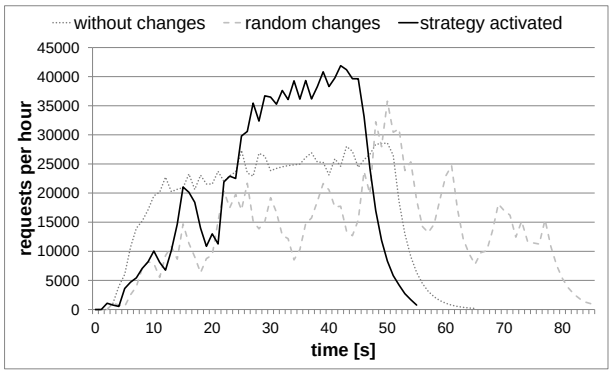

Fig. 6. Request count measured during the experiments

III. 2. Experiments

In order to verify that the strategy meets its requirements, several experiments were performed. The application was installed in an OSGi environment ready to serve requests. Each request followed the numbered sequence highlighted in Figure 4. Three different scenarios were executed:

S1 *without changes* – no interference with the running application[5];

[3] The modeling process, pursuant to the original metamodel, is described in [11] and will not be further elaborated upon here.

[4] in the prototype implementation, the variants' impact and importance of goals are each expressed as a number between 1 and 9 (or *no influence*). Given the variants' impact and current importance of goals it is possible to compare different configurations (if there is more than one possible best configuration, a random one is chosen).

[5] During this scenario the following variants were used: Presentation Server: full content, Search Engine: external, Session Engine: hard disk, Bookings Manager: hard disk, Database: 2 nodes, Ad Server: videos.

S2 *random changes* – with randomization of chosen variants for every service;
S3 *strategy activated* – where the variants were chosen in accordance with the adaptation strategy.

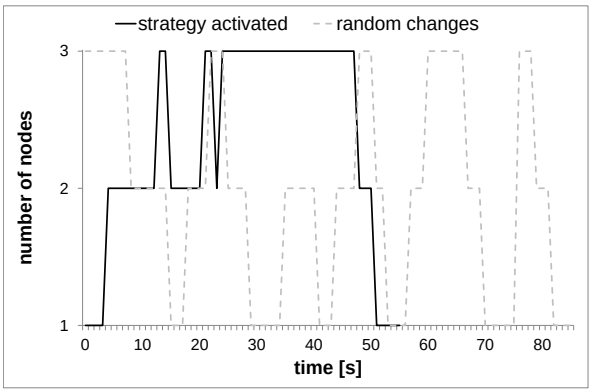

Fig. 7. Chosen variant of the Database service in scenario 2 and 3; in scenario 1 the variant of "2 nodes" was always chosen

During every scenario the same number of clients followed the request flow and requests were dispatched with identical frequency. The request count gradually increased as the experiment progressed, finally reaching a peak and then gradually decreasing all the way down to zero. The response time and the number of requests served per time slot were measured and are shown in Figures 5 and 6. Additionally, the selected Database service variant was monitored, as shown in Figure 7. Since such an experiment includes randomness, it was repeated several times and, as the results were similar, one execution was chosen arbitrarily and is presented in the paper.

III. 3. Discussion

Basing on observations made during the experimentation, we can come to the following conclusions:

1. Figure 6 and 7 together show how the application adjusted to the rising frequency of incoming requests. Once the frequency dropped, the adaptation process switched to using variant *2 nodes* and then *1 node*, thus releasing unneeded resources. Figure 7 shows fluctuation of resources usage in scenarios 2 and 3. Choosing *1 node* and *2 nodes*, especially in period when the request frequency was the highest, caused very long duration of processing all the requests in scenario 2.
2. Following activation of the adaptation strategy the application was able to serve all the clients in a shorter time compared to both remaining scenarios. Scenario S1 was also able to serve all clients but incurred a significantly higher response time.
3. The request count in S3 is lower than in S1 for the first 20 seconds but both scenarios show a similar response time for that period. After 20 seconds the request count in S3 was significantly higher than in S1 and S2.
4. The response time remained short which ensures satisfactory Quality of Experience.

In summary, the results show that when the strategy ensured adequate resource usage. Adaptation brings about two major advantages: 1) response time is kept on a sufficiently low level, and 2) resources are used when necessary and released when no longer needed.

The process shown in this section proved that the presented middleware successfully meets the stated challenges: according to the results, QoS remained high and the application could adapt to the changing context (C5); the declarative way of modeling adaptation (C6) and handling a large number of configurations (C1) as well as validation of configurations prior to installation (C2) remained hidden from the Adaptation Architect; separation of concern was provided by introducing a hierarchical variability model instead of a flat one (C7); the strategy was automatically derived from the adaptation model (C4) and created, modified and (re)installed dynamically at runtime(C3).

IV. RELATED WORK

Adaptivity in SOA systems has been the focus of many research initiatives over the past few years. We have investigated other approaches in order to examine their strong points and conceivably incorporate them into our solution. The authors of [6] present results of their work in the DiVA project which aims at dynamic variability in complex adaptive systems. DiVA proposes a tool-supported methodology for adaptive system design, development and execution. We find its adaptivity description metamodel very convenient and sufficient for our requirements. However, in order to incorporate it into SOA environment we had to extend it as explained in previous sections.

An architecture model called MADAM is presented in [10]. MADAM includes specification of the application structure, its variability and distribution aspects, the properties of each variant, and the utility functions for comparing variants. We describe points of variability using their interfaces. Knowing the interfaces we can dynamically discover components which implement them (considered to represent variants). These components realize parametric adaptation – QoS is tweaked by adjusting certain parameters. For each type of component its ports are distinguished as connection points to other components. Properties of the variants can be expressed as utility functions defined by the Adaptation Architect. Defining such functions can be troublesome with more complex systems. Furthermore, variants are required to possess certain interfaces which cannot change easily at runtime. The DiVA model is free of this drawback.

In [7] self-management of adaptive applications driven by utility functions is presented. The authors claim that utility functions and goal-policies are a good way to describe adaptation because they concentrate on the desired state and not the present state, and they are easy to define. A case study of a resource manager exploiting their concepts is presented. Decisions on how to split the managed resources are made in accordance with declarative utility functions, enabling effective resource distribution. Many such functions are evaluated and their results are compared. However, this approach does not scale well with usage complexity and moreover it lacks a solution that would consolidate many utility functions into an integrated adaptation strategy.

The authors of [8, 15] present a rule-based adaptation framework where adaptation strategies are implemented by certain sets of rules executed by Rule Engines. This approach is sufficient for simple adaptation scenarios; however, when the adaptive application's structure grows, problems with rule complexity and validation emerge.

V. CONCLUSIONS

This paper presents a middleware-supported methodology which addresses QoS adaptation management in the context of SOA systems. The AS3 element pattern was enhanced with a method which creates rules for the Adaptive Manager to execute adaptation strategies built by the Adaptation Architect. Adopting and further extending the DiVA adaptation metamodel enabled the authors to solve many arising problems. Evaluation of the proposed solution demonstrated that the challenges listed at the beginning of the paper are successfully addressed by the created middleware. We therefore believe that our achievement represents a useful and self-contained approach to managing adaptivity of non-functional requirements of SOA applications.

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[2] M. Psiuk, T. Bujok, and K. Zielinski, *Enterprise Service Bus Monitoring Framework for SOA Systems*, IEEE Transactions on Service Computing **5**(3) (2012).

[3] K. Zielinski, T. Szydlo, R. Szymacha, J. Kosinski, J. Kosinska, and M. Jarzab, *Adaptive SOA Solution Stack*, IEEE Transactions on Service Computing **5**(2) (2012).

[4] C. ai Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello, *Modeling and Managing the Variability of Web Service-based Systems*, Journal of Systems and Software **83**(3), 502-516 (2010).

[5] A. Arsanjani, L.-J. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah, *S3: A Service-Oriented Reference Architecture*, IT Professional **9**, 10-17 (2007).

[6] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel, Models in Software Engineering, chapter Modeling and Validating Dynamic Adaptation, pages 97-108 Springer-Verlag, Berlin, Heidelberg, 2009.

[7] J.O. Kephart and R. Das, *Achieving Self-Management via Utility Functions*, IEEE Internet Computing **11**(1), 40-48 (2007).

[8] J. Adamczyk, R. Chojnacki, M. Jarzab, and K. Zielinski, *Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing*, [In:] Marian Bubak, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science – ICCS 2008*, volume 5101 of *Lecture Notes in Computer Science*, pages 355-364 Springer Berlin/Heidelberg, 2008.

[9] T. Nguyen and A. Colman, *A Feature-Oriented Approach for Web Service Customization*, IEEE International Conference on Web Services, 393-400, (2010).

[10] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, *Using Architecture Models for Runtime Adaptability*, IEEE Softw. **23**(2), 62-70 (2006).

[11] S. Dustdar, F. Li, P. Greenwood, R. Chitchyan, D. Ayed, V. Girard-Reydet, F. Fleurey, V. Dehlen, and A. Solberg, *Modelling Service Requirements Variability: The DiVA Way*, [In:] *Service Engineering*, 55-84 Springer, Vienna, 2011.

[12] T. Nguyen, A. Colman, M.A. Talib, and J. Han, *Managing Service Variability: State of the Art and Open Issues*, [In:] *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 165-173, New York, NY, USA, 2011 ACM.

[13] D. Zmuda, M. Psiuk, and K. Zielinski, *Dynamic Monitoring Framework for the SOA Execution Environment*, Procedia CS**1**(1), 125-133, (2010).

[14] M. Psiuk, D. Żmuda, and K. Zieliński, *Distributed OSGi Built Over Message-Oriented Middleware*, Software: Practice and Experience, (2011).

[15] J. Keeney and V. Cahill, *Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework*, IEEE International Workshop on Policies for Distributed Systems and Networks 3, (2003).

Appendix – OUTLINE OF THE DIVA METAMODEL ELEMENTS

To keep the paper self-contained, we briefly outline the original metamodel presented in [6, 11]. The metamodel itself consists of the following elements:

1. Context metamodel – context variables describe the environment around the system in terms of values which should be constantly monitored.
2. Goal metamodel – goals of adaptation (renamed from *properties*) stand for quality marks (reflecting QoS) which should be optimized during adaptation (examples: power consumption, response time, performance). The adaptation goal can be fulfilled by choosing proper variants.
3. Variability metamodel – parameters of adaptation (renamed from *variability dimensions*) and their variants describe adaptation points, i.e. parts of the system which can be managed.The variants are chosen according to their impact on the goals of adaptation and the current context state.

Additional relations bind these three metamodels:

- (1 and 2) Goal management policies (renamed from *property rules*) – used to specify the importance of each goal of adaptation depending on the current context state.
- (1 and 3) Constraints – relationships (dependency, exclusion) between variants and context variables or other variants.
- (2 and 3) Parameters' impact on goals – the way the variants contribute to fulfilling each goal. Impact specifies how each parameter influences each goal of adaptation.

Jacek Psiuk is a PhD candidate at AGH University of Science and Technology in Krakow in a faculty of Computer Science where he receives a doctoral scholarship for his achievements. His research focuses on adaptation of enterprise SOA applications. He conducted research activities in the IT-SOA project. He also participated in the "UniversAAL" project aiming at building an open platform and reference specification for ambient assisted living. As a result he co-authored a paper titled "Seamless Semantic Enrichment of Services in Assistive Environments". Jacek is also a professional software engineer holding a full-time position in the industry. He leads a constantly growing development team employed by Luxoft Poland working for UBS, a Swiss bank. They develop a distributed batch application analyzing and producing finance report data.

Krzysztof Zielinski is a full professor and head of Department of Computer Science at AGH-UST. His interests focus on networking, mobile and wireless systems, service-oriented distributed systems engineering and cloud computing. He is an author of over 200 papers in this area. He served as an expert with Ministry of Science and Education. Now he is leading SOA-oriented research performed by IT-SOA Consortium in Poland. In this area his research interest concerns: Adaptive SOA Solution Stack, Services Composition, Service Delivery Platforms and Methodology. He is a member of IEEE, ACM and Committee on Informatics of Polish Academy of Sciences. He served as a program committee member, chairman and organizer of several international conferences including MobiSys, ICCS, ICWS, IEEE SCC and many others.

DOI:10.12921/cmst.2015.21.01.001