Authors copy (Camera ready version). Final version will be accessible on LNCS site.

Service Variability Patterns

Ateeq Khan[1], Christian Kästner[2], Veit Köppen[1], and Gunter Saake[1]

[1] University of Magdeburg, Germany.
[2] Philipps Universität Marburg, Germany.
ateeq,vkoeppen,saake@iti.cs.uni-magdeburg.de
kaestner@informatik.uni-marburg.de

Abstract. Service-oriented computing (SOC) increases flexibility of IT systems and helps enterprises to meet their changing needs. Different methods address changing requirements in service-oriented environment. Many solutions exist to address variability, however, each solution is tailored to a specific problem, e.g. at one specific layer in SOC. We survey variability mechanisms from literature and summarize solutions, consequences, and possible combinations in a pattern catalogue. Based on the pattern catalogue, we compare different variability patterns and their combinations. Our catalogue helps to choose an appropriate technique for the variability problem at hand and illustrates its consequences in SOC.

1 Introduction

Service-Oriented Computing (SOC) is a paradigm to create information systems and provides flexibility, interoperability, cost effectiveness, and higher quality characteristics [1]. The trend of service-usage is increasing in enterprise to support processes.

However, even in the flexible world of services, *variability* is paramount at all layers. Variability is the ability of a system to extend functionality, modify, customise or configure the system [2]. We do not want to provide the same service to all consumers but need to provide customised variants. Consumers want to fine tune services according to their needs and will get a unique behaviour, which is tailored (personalised) for their requirements. Fine-tuning depends on available *features* of the services, where a feature is a domain-abstraction used to describe commonalities and differences [3].

However, variability approaches in SOC are ad-hoc. Many solutions exist; however, each one is tailored and aimed for a specific problem or at a specific layer. Some approaches use simple mechanisms for variability, such as, using if-else structure implementations for variability in services. Others try to prevent bloated results of putting all variability into one service (which also violates the service principle that each service should be an atomic unit to perform a specific task) with various strategies, such as frameworks ([4–6]) and languages-based approaches ([7, 8]). A single and perfect-for-all solution does not exist in variability. Such a solution is also unrealistic, due to very different requirements and technologies at different layers. Still, we believe that there are common patterns, and developers do not need to rule out inefficient solutions and reinvent better solutions again and again.

We contribute a catalogue of common variability pattern, designed to help developers to choose a technique for specific variability needs. We survey the literature and abstract from reoccurring problems and individual implementation strategies and layers. We

summarise our results in six common patterns for variability in the SOC domain (in general many patterns are even transferable to other domains). The patterns are general enough to describe the problem and the solution strategy including its trade-offs, different implementation strategies at different SOC layers, but also concrete enough to guide a specific implementation. To help developers decide for the appropriate solution to a variability problem at hand, we discuss trade-offs, limitations and possible combinations of different patterns. To aid understanding, we discuss example scenarios of each pattern with their consequences.

2 Variability Patterns in SOC

We use the pattern template by Gamma et al. [9] with modification to describe our patterns in SOC domain. Our pattern structure is simple and consists of a pattern name, pattern motivation or recurring problem, applications, examples, implementation technique or solution for the pattern, and consequences of the pattern.

In our pattern catalogue, we include some general patterns which are used in various variability situations. We discuss some implementation techniques and examples. Discussion of all implementation techniques for each pattern is out of scope of this paper (for details see [10]). In contrast to related pattern catalogues [2, 11], which are focused on product lines, we focus on the SOC domain. We use examples from a sports SaaS application, which is used to manage a sports club. The SaaS application contains different services, e.g. to display the matches' results, managing players and members. Our sports application can be used in other sports domains by using variability approaches.

2.1 Parameter Pattern

Motivation: Service providers offer different implementations of a service and selection of services are based on the parameters. Service consumers have different kinds of preferences for a service or need a specific behaviour from services.
Application: This is a simple and widely used pattern. This pattern provides variability solutions based on parameters. Service providers offer variability depending on parameters (depicted in Figure 1) e.g. who is calling the service (consumer id). Access to specific services is decided using this pattern. Service providers plan for variability at design time and this result in variability for a consumer at runtime. Parameters and consumer specific configurations may be bundled together and stored. There are different options to store the configuration of the consumers, mostly stored at the service provider side (although, storage does not have an impact, e.g. at the consumer side or at the service provider side). When a consumer accesses the service, consumer specific configuration is accessed for variability and unique behaviour. We can use parameter pattern for user-interface or workflow preferences, database attributes or for domain specific extensions.
Example: We can use the parameter pattern for sorting, rendering, or for different layouts in a sports service scenario, e.g. offering text commentary of a match based on the consumer language or changing scoring fields for different sports domain.
Solution: We can store consumer specific settings and parameters as configuration files, e.g. as XML files or stored in a database for each consumer. Parameter storage is also not necessary; a possible extension is passing all required parameters every time when a consumer accesses the SaaS application. There are two types of data associated with this

pattern, one is configuration specific data (values configured by consumers for different options) and other is application specific data for each consumer (contain database, values, and users). Configuration data is usually small and less updated as compared to application specific data. For general needs or requirements, configuration data for each consumer can be stored as key-value pair, e.g. consumer id and configuration values (for user-interface favourite colour, selected endpoint, or fields to display).
Consequences: This pattern provides an easy approach to provide variability from the same source code by storing and accessing consumer-specific behaviour based on parameters. Services are selected based on attribute values. Such approach is simple to program and does not require a lot of expertise. This pattern provides flexibility but consumer can choose only from the provided set. Management will be an issue in larger scenarios if parameter conditions are scattered within the code.

2.2 Routing Pattern

Motivation: Even if requirements are same between two consumers, business rules can vary between them. Consumers want to change the business rules to follow a specific behaviour. This pattern routes the request based on the rules or consumers requirements.
Application: We can use this pattern for routing requests to different targets, selection of services, changing application behaviour using rules or based on consumer description. Changes can be made at runtime. Flexibility is provided by consumer, provider or by both depending on the scenario and can be used at different layers. Service providers offer consumers to change the business rules of an application. Rules are used to handle complex scenarios and different conditions. Such conditions are due to user preferences. Meta rules or algorithms can be used to choose which rule has to be executed. Service providers can also allow to use specific operators, e.g. allowing consumers to add if-else branches in the business rules (shown in Figure 2) to control the business logic or using logical operators. Logical operators can also be source of variability, e.g. some consumers may use simple operators and others prefer or require more flexible rules for business logic. We can use this pattern to to handle exceptions. This pattern is similar to the façade or proxy pattern, discussed in [9].
Example: In our sports system, members pay club membership fees. For payments different options, or routing of services are possible, e.g. local members pay using credit card, bank transfer or both, and foreign members can only pay using credit card.

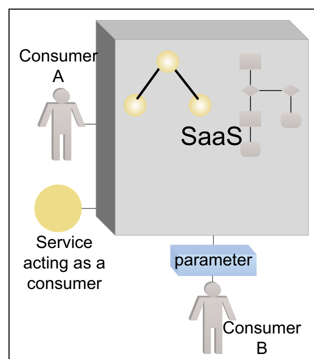

Figure 1: Parameter pattern

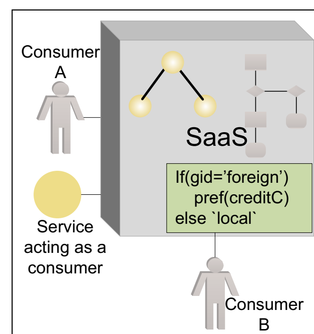

Figure 2: Routing Pattern

Solution: Different solutions for implementation do exist for routing. These approaches range from simple if-else statements to complex Aspect-Oriented Programming (AOP) based approaches [12]. Message interception can also be used for routing. A message is intercepted and analysed to add user-specific behaviour. Different techniques are used to intercept message. Rahman et al. [12] use an AOP-based approach to apply business rules on the intercepted message in SOA domain.

A web service request is intercepted, rules are applied on the request, and then result is forwarded. Routing can be done by analysing SOAP header or SOAP body (may carry extra data for routing) and request is routed accordingly.
Consequences: Routing allows consumer to use application which suits to their requirements. It also allows to separate business logic from service implementation (for easy modification in rules at runtime). It is also easy to change routing rules and only few changes are necessary. Consumers influence the application behaviour by changing rules.

Adding new business rules or logical operators may add unnecessary loop in an application or inconsistency in application. Validation rules or validators are applied before adding branching rule [13, 14]. Higher complexity of involved services may lead to inconsistency in application due to rules. Algorithms for validation [15] can also be used to find inconsistent or contradictory rules. Scalability is also an issue for complex applications, routing rules may increase in size and their management become difficult. Routing pattern may introduce single point of failure or decrease in performance.

2.3 Service Wrapping Pattern

Motivation: We use this pattern when service is incompatible to use (due to technical or business issue) or provider want to add/hide functionality in services. So, modification is required to use the service in a scenario.
Application: We can use this pattern (as depicted in Figure 3) for the wide variety of changes, e.g. from technical perspective interface mismatch, message or data transformation, protocols transformation, or for business modifications (content modification). This pattern helps to delegate, modify or extend the functionality for consumers [11, 16]. Service wrapping can be used to modify existing services and to resolve incompatibilities between services or service interfaces. Services are wrapped and arranged together so that a service delegates the request to other services or component, which implement the service logic. Composite, decorator, wrapper, proxy, and adaptor patterns [9] are similar patterns with the service wrapping pattern. We can also use this pattern to offer a group of services (from different providers, platforms, or languages) as a composite service to provide sophisticated functionality and vice versa. Consumers use services through the provided interface without knowing whether the service provider adds or hides the functionality. We can also use this pattern to support legacy systems without major modification of existing code of the system and exposing functionality as a service [1, 17, 18]. The consumers may want to expose her existing systems as a service for other consumers, and restrict the access of some private business logic.
Example: An example from our sports system is offering email and SMS message services (wrapped together as a *notify match* composite service) to send reminder about change in match schedule to members and players.
Solution: We can use different solutions, e.g. using intermediate service, middleware solutions or tools for variability. To expose legacy systems as service, different techniques

are possible, e.g. service annotations in Java. Intermediate service acts as an interface between incompatible services and contains required logic to overcome the mismatch.

Using SAP Process Integration (SAP PI) as a middleware, different service implementation, workflows, or client interfaces can be used to provide variability. We can use different types of adapters to solve interface mismatch or to connect different systems. When a request from a consumer side is sent to SAP PI, different service implementations, business rules, and interfaces can be selected based on the request. We also use middleware for synchronous-asynchronous communication, in which results are stored at middleware and delivered to the consumers based on their requests. The consumer or provider both (not necessary service owner, could be third party providers) are responsible for variability in this pattern.

Consequences: Using this pattern, we offer different variability solutions. Service wrapping hides the complexity of the scenario from the consumer and simplifies the communication between consumer and composite service (consumers do not care about different interfaces or number of underlying services). Addition or removal of a service becomes easy for the consumer (considered as include/exclude component in case of component engineering). Services are reused and become compatible without changing their implementation details by using service wrapping.

Composite services increase the complexity of the system. Adding services from other providers may effect non-functional properties. Service wrapping increases the number of services (depending on the scenarios composite, adapters or fine-grained) offered from the provider and management of such a system becomes complex.

2.4 Variant/Template Pattern

Motivation: We assume that providers know consumers variability requirements for services. Therefore, providers offer static variants of services, and consumers configure these variants according to their needs, e.g. variants based on the consumer geographical location, cultural aspects, subscription, consumer group, and devices.

Application: Providers offer a set of service variants to consumers (as illustrated in Figure 4). Service providers plan for the variability and provide variants at design time and consumers select these variants, mostly at runtime. Service providers select features and varying options based on industry best practices, as variants, with a pre-defined set

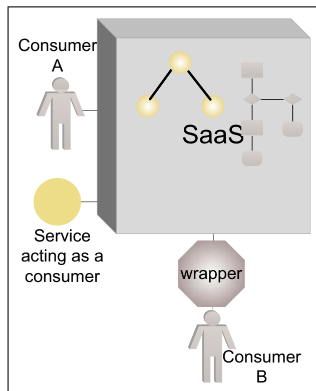

Figure 3: Service Wrapping Pattern

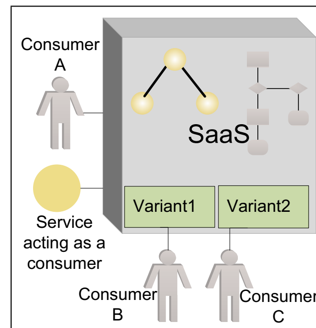

Figure 4: Variant Pattern

of configuration options. Consumers choose options. In [13, 14], authors suggest to offer a set of templates, so consumers can choose a template in a process or workflow. In [19], authors discuss different workflow patterns.
Example: In our sports system, different user-interface variants are used to display match scores (suppose in Figure 4, text commentary is displayed in *Variant1* for the consumer B, online video streaming in *Variant2* from provider side for consumer C, while the consumer A sees the score only).
Solution: The variant pattern is a general pattern and used in various scenarios. Consumers choose from a set of variants and use options to configure it, e.g. for unique look and feel, workflows, or for viewing/hiding data fields in interface. These variants are typically generated from the same source code at provider side. We can use generators, inheritance, polymorphic, or product line approaches to generate variants of a service at design time [3, 9, 20, 21]. In [3], we discuss how different variants can be offered based on a feature set from the same code base and benefits achieved using variability. WSDL files can also be tailored and used for representing different variants. The consumer specific options are stored as configuration files.
Consequences: It pattern allows to offer optimal solutions in the form of variants. Industry best practices help consumers to choose right options and result in higher quality.

This pattern does not allow full flexibility to consumers. Developers provide variants in advance and consumers have to choose only from given set. Managing different variants of a service increases the complexity. Additional information is needed to decide which variant of a service is useful or compatible. Complex scenarios need a flexible platform or architecture, which allows handling of different variants (challenges mentioned in [3]).

2.5 Extension Points Pattern

Motivation: Sometimes, consumers have specific requirements which are not fulfilled by the above mentioned patterns. For instance, consumers want to upload their own implementation of a service, replace part of a process, to meet the specific requirements. Therefore, providers offer extension points in a SaaS application.
Application: This pattern requires pre-planning. Service providers prepare the variability as extension points at design time. Consumers share the same code base and provide behaviour at those extension points at runtime. Other consumers access the service without any change. It is similar to the strategy design pattern [9], frameworks, or callbacks (can use inheritance methods at design time). The consumer modifies the application behaviour by uploading implementations, rules, or fine-tuning services (changing service endpoints). Extension points allow consumers to add consumer-specific implementations or business logic in the system at runtime as shown in Figure 5.
Example: In our sports system, a consumer configures extension point for alternative scoring services from different providers using web service endpoint binding method.
Solution: In SOC, service interfaces (WSDL files), service implementations, service bindings, and ports (endpoints) act as extension points in the architecture [22–24]. Consumers change these extensions points for variability. We can use physical separation of instances or virtualisation as solutions for this pattern. A provider allocates a dedicated hardware or a virtual instance for consumer-specific code execution separately. In case of malicious code or failure, only the tenant-specific instance or virtual image will be

effected instead of the whole system. The consumer can perform modifications for service binding in WSDL. Endpoint modification is a method to modify the service address in a WSDL or in a composite service, e.g. adding an end-point service as an alternative in a web service binding. Endpoint modification can be done at runtime.
Consequences: Extension points offer flexibility to the consumer and allow customisation of application behaviour. There are some potential risks due to offering flexibility through extension points. In a workflow, by allowing a consumer to add activities, it is possible that adding new activities in a workflow introduce loops in application, consuming resources or might result in never ending loops. Another problem is in allowing a consumer to insert her own code, which may lead to failure of the whole system or instance, e.g. in case of malicious code or virus uploading. Once variability is realised by consumers, the system must check for the modification (extension points) and test scenarios for correctness of the system, e.g. for resource consumption or effect on the whole process (availability, time constraints for response, etc.)

2.6 Copy and Adapt Pattern

Motivation: Offering variability from the same code base in SaaS is not always a best choice. Sometimes, available patterns or approaches fail to fulfil consumers demands from the same code base. Another reason is, if we apply those patterns, management become complex or result in higher costs as compared to separate service instances.
Application: We use this pattern when shared instance modifications for a consumer harm other consumers. Therefore, a developer copies the service code and modifies it for individual consumer as depicted in Figure 6. This pattern requires source code access for modification. Mostly, the consumer is responsible for managing changes or updating the new version of service with own modifications. We also use this pattern where consumers have data privacy issues, e.g. in some countries, data storing, or processing in the shared environment is not feasible.
Example: We use this pattern in scoring service. Scoring is different for football (for consumer group A) as compared to baseball (for consumer group B) and a lot of changes are required, which makes the scenario complex.
Solution: Service providers offer a separate instance for a consumer to keep the solution simpler, although it may introduces services with similar codes and functionalities. The consumer introduces her own implementation and exposes as a service or modifies

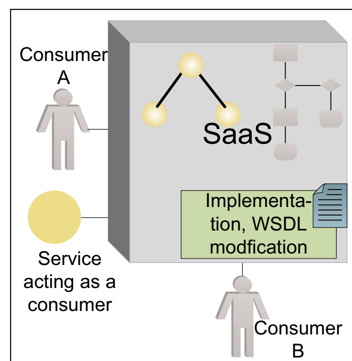

Figure 5: Extension Points Pattern

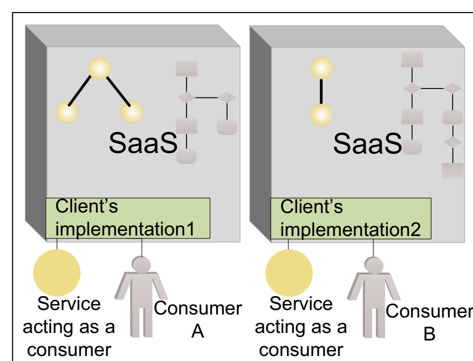

Figure 6: Copy and Adapt Pattern

the provided solution. In such a case, every consumer gets a independent customised service instance. We use this pattern at the process or database layer as well, where a consumer adds or develops her own process in SaaS. In such cases, at the database layer, a consumer uses a separate database instance to accommodate new database relations and different business requirements.

Consequences: SOC benefits are achieved in this pattern, although for some parts the application service provider (ASP [25]) model is used, in which each consumer shares the infrastructure facilities (shifting infrastructure and management tasks to providers) but separate service instances. Legacy systems or other applications can be shifted to SOC using this pattern easily. This pattern allows full flexibility, and consumers can modify or customise respective services freely.

From service provider perspective, this pattern does not scale. It is expensive in terms of costs for the large number of consumers and service instances. Hardware costs also increase in such cases due to separate instances. Code replication increases the effort for management and decreases productivity. Software updates or new version of software must be updated for each instance manually or individually. Due to these main problems, it is often not advisable to use this pattern and sometimes considered as anti-pattern.

3 Patterns Comparison and Combinations

Patterns	Required changes	Flexibility	Scalability	Risk	Maintenance	Responsibility
Parameters (P)	low	medium	high	low	easy	provider
Routing (R)	low	medium	medium	medium	easy	both
Service Wrapping (SW)	medium	high	medium	medium	medium	both
Variants (V)	very low	low	medium	low	medium	provider
Extension Points (E)	medium	medium	low	high	difficult	provider
Copy and Adapt (CA)	very high	very high	high	low	difficult	both
Combining P + E	medium	medium	high	low	low	provider
Combining R + SW	medium	high	high	low	medium	consumer
Combining R + V	low	high	medium	low	medium	both
Combining R + E	medium	low	low	medium	difficult	both
Combining SW + V	medium	medium	high	low	medium	provider
Combining V+ E	medium	high	medium	low	medium	provider

Table 1: Pattern comparison and combinations

We discuss different patterns for variability in SOC. In Table 1, we compare these patterns with each other against evaluation factors for variability. Our pattern catalogue covers the common variability problems and solutions in SOC and by no means a comprehensive pattern catalogue. We identify that some patterns can also be combined together for a better solution or to solve variability problems. For example, the *parameter pattern* can be combined with the *extension points pattern* to keep the consumer implementation separate from other consumers. Consumer's implementations are stored in configuration files and retrieved when consumers access the service.

We can also combine the *routing pattern* with the *variant pattern* or the *service wrapping pattern* to select different variants of services and protocols or messages transformation based on some criteria. The *routing pattern* is used with the *extension points pattern* to inject routing rules in application (e.g. uploading code containing

routing logic). We can also use the *routing pattern* to offer a set of valid rules based on variants. The *service wrapping pattern* can be mixed with the *variant pattern* or the *routing pattern* to offer different variants of services. Theses variants are shared between consumers and used for different service flows or to overcome a mismatch at middleware level. The *bariant pattern* with the *extension points pattern* allows us to restrict the extension points options to valid combinations instead of giving consumers flexibility to add random activities. So, consumers can add activities or rules from offered templates. An example of such an activity in our sports system is a *notification activity* where a consumer can send an email for a match notification but other consumers want to add additional SMS message activity for notification. So, SMS message activity can be added in the workflow from templates activity.

It is possible that different patterns fit in a particular environment or problem. Choosing a pattern depends on many factors, e.g. patterns consequences, application scenarios, business needs, architectures, and customers business models. In some organisation and countries, consumers have legal or organisational issues, restrictions for shared access of applications (despite the efforts for data and processes confidentiality in multi-tenant applications), so the consumer may prefers other patterns.

4 Summary and Outlook

We contributed six variability patterns for SOC that can guide developers to solve different variability problems in practice. We discuss trade-offs according to several evaluation criteria to help deciding for the right solution strategy for a problem at hand. Our pattern catalogue helps to reuse solutions strategies in a manageable way.

In future work, we plan to extend our pattern catalogue into a framework that contains decision criteria to choose and manage variability in SOC with specific implementation techniques. We will also evaluate our pattern catalogue further in practice to compare performances where more than one patterns can be used at the same time.

Acknowledgement

Ateeq Khan is supported by a grant from the federal state of Saxony-Anhalt in Germany. This work is partially supported by the German Ministry of Education and Science (BMBF), within the ViERforES-II project No. 01IM10002B.

References

[1] Papazoglou, M.P., van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. VLDB **16**(3) (2007) 389–415
[2] Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software - Practice and Experience **35**(8) (2005) 705–754
[3] Apel, S., Kästner, C., Lengauer, C.: Research challenges in the tension between features and services. In: ICSE Workshop Proceedings SDSOA, NY, USA, ACM (May 2008) 53–58
[4] Cámara, J., Canal, C., Cubo, J., Murillo, J.M.: An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. Electr. Notes Theor. Comput. Sci **189** (2007) 21–34
[5] Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A framework for native multi-tenancy application development and management. The 9th IEEE International Conference on E-Commerce Technology (2007) 551–558

[6] Kongdenfha, W., Saint-Paul, R., Benatallah, B., Casati, F.: An aspect-oriented framework for service adaptation. In: ICSOC. Volume 4294 of LNCS., Springer (2006) 15–26

[7] Charfi, A., Mezini, M.: AO4BPEL: An aspect-oriented extension to BPEL. In: WWW **10**(3) (2007) 309–344

[8] zur Muehlen, M., Indulska, M.: Modeling languages for business processes and business rules: A representational analysis. Information Systems **35** (2010) 379–390

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts (1995)

[10] Khan, A., Kästner, C., Köppen, V., Saake, G.: Service variability patterns in SOC. Technical Report 05, School of Computer Science, University of Magdeburg, Magdeburg, Germany (May 2011) http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/KKKS11.pdf.

[11] Topaloglu, N.Y., Capilla, R.: Modeling the variability of web services from a pattern point of view. In Zhang, L.J., ed.: Proceedings of the European Conference on Web Services ECOWS. Volume 3250 of LNCS., Springer (2004) 128–138

[12] ur Rahman, S.S., Khan, A., Saake, G.: Rulespect: Language-Independent Rule-Based AOP Model for Adaptable Context-Sensitive Web Services. In: 36th Conference on Current Trends in Theory and Practice of Computer Science (Student Research Forum). Volume II., Institute of Computer Science AS CR, Prague (January 2010) 87–99

[13] Chong, F.T., Carraro, G.: Architecture strategies for catching the long tail. http://msdn.microsoft.com/en-us/library/aa479069.aspx last accessed 24.06.2011 (April 2006) Microsoft Corporation.

[14] Carraro, G., Chong, F.T.: Software as a service (SaaS): An enterprise perspective. http://msdn.microsoft.com/en-us/library/aa905332.aspx last accessed 24.06.2011 (October 2006) Microsoft Corporation.

[15] Bianculli, D., Ghezzi, C.: Towards a methodology for lifelong validation of service compositions. In: Proceedings of the 2nd international workshop on Systems development in SOA environments. SDSOA, New York, NY, USA, ACM (2008) 7–12

[16] Mügge, H., Rho, T., Speicher, D., Bihler, P., Cremers, A.B.: Programming for Context-based Adaptability: Lessons learned about OOP, SOA, and AOP. In: KiVS 2007 - Kommunikation in Verteilten Systemen - 15. ITG/GI-Fachtagung. (2007)

[17] Yu, Q., Liu, X., Bouguettaya, A., Medjahed, B.: Deploying and managing web services: issues, solutions, and directions. The VLDB Journal **17**(3) (2006) 537–572

[18] Mughrabi, H.: Applying SOA to an ecommerce system. http://www2.imm.dtu.dk/pubdb/p.php?5496 last accessed 05.05.2011 (2007) Master thesis.

[19] Aalst, W., Hofstede, A., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51

[20] Papazoglou, M.P., Kratz, B.: Web services technology in support of business transactions. Service Oriented Computing and Applications **1**(1) (March 2007) 51–63

[21] Pohl, C., Rummler, A., et al.: Survey of existing implementation techniques with respect to their support for the requirements identified in m3. 2 (July 2007) AMPLE (Aspect-Oriented, Model-Driven, Product Line Engineering), Specific Targeted Research Project: IST- 33710.

[22] Jiang, J., Ruokonen, A., Systa, T.: Pattern-based variability management in web service development. In: ECOWS '05: Proceedings of the Third European Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2005) 83

[23] Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: WWW, ACM (2008) 815–824

[24] Erradi, A., Maheshwari, P., Tosic, V.: Policy-driven middleware for self-adaptation of web services compositions. In Middleware 2006. Volume 4290 of LNCS. Springer (2006) 62–80

[25] Lacity, M.C., Hirschheim, R.A.: Information Systems Outsourcing; Myths, Metaphors, and Realities. John Wiley & Sons, Inc. (1993)