Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1307–1314 DOI: 10.15439/2017F397 ISSN 2300-5963 ACSIS, Vol. 11

Aspect-driven Context-aware Services

Karel Cemus, Filip Klimes
Dept. of Computer Science
Czech Technical University in Prague
Prague 2, 121 35, Czech Republic
Email: {cemuskar,klimefi1}@fel.cvut.cz

Tomas Cerny
Dept. of Computer Science
Baylor University
Waco, TX, 76798, USA
Email: tomas_cerny@baylor.edu

***Abstract*—Nowadays enterprise software solutions must deal with ever-growing complexity and a multitude of business processes. The mainstream system design decomposes the system into small reusable services. While these services isolate certain system logic and address efficient elasticity towards growing user demands, there are multiple issues related to such a design, such as limitations to deal with restated information, information reuse or the ability to address cross-cutting concerns across multiple services. This paper highlights limitations of service-oriented architecture and proposes an alternative decomposition through aspect-driven service-oriented architecture. Such architecture involves adaptive, context-aware services preserving simple maintenance while addressing information reuse and crosscuts across services. The paper provides a formal description of the proposed architecture as well as a demonstration through a case study, showing approach properties and benefits.**

I. Introduction

CONTEMPORARY Enterprise Information Systems (EISs) grow in both scale and complexity. Functional requirements are becoming more advanced because they require context-awareness. Considering various aspects of a business domain within current execution context, i.e., within time, user's privileges, and state of the system. Non-functional requirements often include scalability and distribution, handle and serve a large amount of requests or process large volumes of data [1].

Having this mind, conventional systems have to deal with several aspects of a business domain. Besides complex domain models [2], there are access policies and business rules that need to be implemented to properly secure and maintain data.

For illustration, consider a basic e-shop system as an example of conventional EIS. There are *users* representing both customers and employees with various access roles and responsibilities. Next, there are *products* with description and photo gallery, organized into categories, and connected to the *store* to manage delivery and stock. Finally, the system maintains *orders*, their state, changes in time, billing, and state of delivery. Obviously, even this simple and reduced example is quite complex. The business model is tangled and there are a few stateful objects implying conditional business rules and access policy. Finally, there exist 3rd party services for *billing*, *shipping*, and *emailing*. For the sake of simplicity, we do not consider sales introducing time-based conditions combined with the stock.

One common way to implement these systems is to use conventional technologies that head towards monolith applications with poor scalability and maintenance. This results from difficult domain decomposition and high information repetition due to significant concerns tangling [3]. Alternatively and more likely, to deliver a highly scalable and distributed system, there exists Service-Oriented Architecture (SOA) [1], [4], [5] decomposing a system into many smaller services following Single Responsibility Principle [6]. However, while this decomposition increases scalability and throughput, its maintenance gets more difficult as the services are more encapsulated, self-standing, isolated, and possibly written in different programming languages, which significantly reduces a possibility of information reuse and forces manual repetition instead.

In this paper, we discuss system decomposition into services, and highlight limitation of the overall architecture. Next, we propose an enhancement of SOA through adaptive context-aware services preserving simple maintenance and keeping minimal information restatement.

This paper is structured as follows. In Section II, we discuss SOA more deeply and in Section III highlight its limitations and opened challenges. In Section IV, we present Aspect-Oriented Design Approach efficiently dealing with business rules repetition and transformation, and we generalize and modify this approach to fit SOA environment and present the design in Section V. We show a case study in Section VI and in Section VII we elaborate and briefly evaluate the alternative existing approaches. We conclude the paper in Section VIII.

II. Conventional Design

Complexity and wide use of EISs emphasize their robustness and scalability. The common approach in SOA to design a large distributed system suggests decomposition of the application logic into small encapsulated standalone units called *services* responsible for and encapsulating a part of the business domain [4]. For example, in the e-shop system, one service is responsible for the user management, while the other for the product management. These services are then composed together to deliver more complex functionality. An example of such a *composite service* [7] is the orders management. It depends on both users and products plus adds additional features.

Definition. *A service is a reusable, cohesive, managed, deployable, and independent process interacting via messages. [7] [8]*

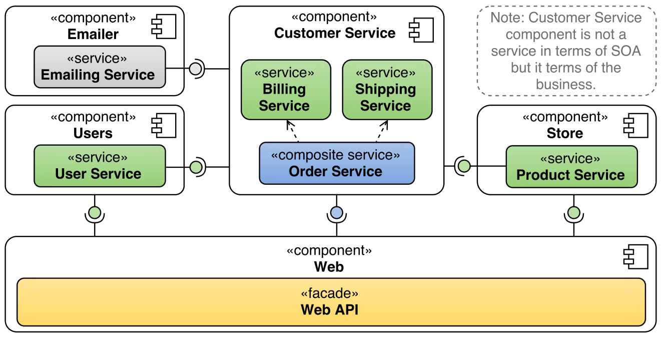

«component» Emailer
«service» Emailing Service
«component» Customer Service
«service» Billing Service
«service» Shipping Service
«composite service» Order Service
Note: Customer Service component is not a service in terms of SOA but it terms of the business.
«component» Users
«service» User Service
«component» Store
«service» Product Service
«component» Web
«facade» Web API

Fig. 1: E-shop design in conventional SOA

Definition. *A composite service accesses and combines information and functions from existing service providers. [7]*

SOA suggests a system infrastructure following the structure of the real business [4]. In consequence, when the e-shop has departments responsible for the store (products and stock) and customer service (orders), then these are also components [1], [7] in the system. The internal design of these components may differ based on their implementation. Properly implemented SOA emphasizes scalability and testability because it significantly reduces the complexity of services comparing to monolith applications [8]. On the other hand, it significantly reduces and complicates information reuse and enforces manual repetition instead, which easily leads to inconsistencies [3] and error-prone and expensive maintenance.

Definition. *SOA is set of design principles organizing software components (services) around business capabilities and connects them through standard interfaces and messaging protocols. Each component is self-contained, black box for consumers, and exposes only its interface [7], [9].*

Having the e-shop example from the previous section, Figure 1 presents the system architecture applying conventional SOA, more specifically Microservices[1] pattern. The *User*, *Store*, and *Customer service* components represent departments of the business with a *Web* service as a composite service implementing a user interface. The services interact through a RESTful protocol[2] [11]. We use this example as a reference later in this paper.

III. Challenges in Service-Oriented Architecture

Decomposition of a system into these small units delivers much simpler design, evolution, and maintenance of both units and the system itself. Furthermore, the communication through a neutral environment such as HTTP protocol makes the services independent of a particular technology. That enables us to develop each service in a different programming language and to use different frameworks. Next, the natural decomposition by the business structure clearly defines responsibilities of components, which supports agile programming [1], multi-team development, and rapid delivery. Unfortunately, there still remain some challenges we face. Among others, we address the following issues in this paper.

A. Composition of Business Rules

First, while decomposition brings clear design, the composition requires reuse of information from services it depends on. For example, *Order* composite service needs to know the model structure and the business rules of the underlying services *User* and *Product* to be able to validate incoming orders or even additionally transform and expose the rules to web API, e.g., for client-side validation necessary for a user-friendly user interface. However, distributed environment and possibly different technologies basically prevent the simple sharing. The model description can be exposed through API schema[3], but reuse of business rules is very difficult even in monolithic applications [3], [14].

B. Business Domain Configuration

Business domain configuration is a special case of service composition. In an existing system, multiple services often need to share some configuration, for example, a VAT percentage or business hours definition. While the VAT computation could be extracted to a single specialized microservice, a specialized service determining whether now are business hours or not seems to be unnecessary overhead. Instead, shared simple configuration would be the much easier solution. Unfortunately, either we configure each service separately and have difficult maintenance due to information restatement, or we basically hit a special case of service composition; this is similar the reuse of business rules of a basic service configuring the domain. Neither way it is efficient and easily maintainable using conventional technologies.

C. Business Rules Maintenance

Having the business logic distributed into many self-standing services carries besides benefits also difficult evolution. When a change request occurs, we must manually update each affected service. The effort might be too high to make a small change such as an adjustment of business rules due to changes in the business domain. Unfortunately, there is no way to share the rules or update them in a batch.

D. Business Documentation Extraction

Finally, the overall system can get quite complex especially when the system is large or grows. Acquisition of current business documentation covering the services, their operations, model, and applied business rules is very challenging and often requires a lot of manual efforts. Extraction of this information from distributed systems is very limited.

[1]There exist attempts to deprecate SOA due to poor implementation and many failed projects in past. However, the approach itself is general and there are more evolved specializations such as Microservices pattern [9].

[2]Microservices pattern suggests a use of a simple connection and smart endpoints [9], but we could also use more complex alternatives such as catalogs, service locators, and Enterprise Service Bus [5], [10]. It would be major overhead in this example plus conventional approaches highlight choreography over orchestration [8].

[3]We may use SOAP with WSDL [12] or RESTful services optionally with controversial WADL [13].

In this paper, we present a novel adjusted aspect-driven design approach to fit SOA and distributed systems. The approach focuses on simplification of development and maintenance through the elimination of manual information repetition. Instead, it automates transformation, reuse, and restatement of business rules, which enables us to provide an alternative and efficient solution to these challenges.

IV. Aspect-Driven Design Approach

As we demonstrated in the previous sections, there are concerns in EISs, which are hard to effectively capture within SOA, e.g. business rules composition and maintenance. We call them *cross-cutting concerns* because they affect other concerns throughout the system. For instance, multiple services are affected by the underlying data model, or they are subject to a global business rule. By using conventional approach, these concerns usually get tangled into the underlying code in multiple points, making it hard to develop and maintain.

Definition. *A cross-cutting concern, or aspect, is a system property which affects other system components by cross-cutting their functionality. [15]*

Aspect-Driven Design Approach (ADDA) utilizes principles of Aspect-Oriented Programming [15] (AOP) to tackle problems introduced by cross-cutting concerns in monolithic EISs. It reduces information restatement through extraction of tangled concerns and their isolation in the single focal point [3]. The concerns are then automatically distributed throughout the system by aspect weavers at runtime. This leads to more efficient development and maintenance of such system, as well as it reduces the risk of human error, compared to manually repeated and tangled concerns. Furthermore, the concern distribution can be carried out across different platforms [16]. This helps us to use various technologies for individual modules while preserving the single point of truth.

ADDA uses Domain-Specific Languages (DSLs) rather than General Programming Languages to capture the cross-cutting concerns. DSLs are more efficient in describing domain-specific logic, as they are tailored for that particular domain while relaxing stress on generality. This reduces development efforts and enables domain experts to directly participate in the system development [17].

In ADDA, EIS is perceived as a multi-dimensional space [3], with the concerns as individual axes and the states of the system as points in such a space. The state of the system is determined by its current *execution context* and *business context* [18], e.g., a locale of the user, a business operation, and the current time. Based on the information from the current context, respective concerns are dynamically weaved together at runtime.

Definition. *The Execution context is a complex information structure including information about the current Application context, User context, and operation parameters. [18]*

Definition. *The Application context of EIS is a set of global variables and their values at the current point in time. [18]*

Definition. *The User context of EIS is a set of information about the current user of the system. [18]*

Definition. *The Business context is a set of preconditions and post-conditions defined by a business operation. [18]*

As we have established, ADDA simplifies separation of cross-cutting concerns through their description in DSLs and automated transformation and distribution from the single point of truth. Therefore, it reduces maintenance efforts through reduction of manual information restatement. However, this comes with a significant cost of initial investment, as the weavers and DSLs need to be implemented first. They are not project-specific and can be reused, though.

V. Aspect-Driven Service-Oriented Architecture

In SOA, composite services face to the challenge of limited inspection of business contexts, i.e., limited reuse of business rules declared by services they depend on. It results from difficult information extraction. In this chapter, we introduce modified service design to ease information inspection and exposition. That enables us to define all the information in the single point of truth and then automatically transform, reuse, and distribute it at runtime. Next, with runtime composition, we are able to consider current execution context and thus make the services context-aware. In order to apply AOP-based principles, we identify the cross-cutting concerns, i.e., the aspects, and formalize the challenge in terms of AOP.

Note. *In this section, we demonstrate the concept on the e-shop system example described in Section II.*

A. Formalization

First, we identify the *aspects* in the system:

(i) *Business context* of a business operation defines *business rules* and *business domain configuration*. Operations of composite services often reference business contexts, or their subsets, from services they depend on. Consider the Order Service. For example, order creation validates the input also by the rules specified by the user creation operation in the User service. It also references business domain configuration of the Billing service, e.g., VAT percentage to properly compute the price.

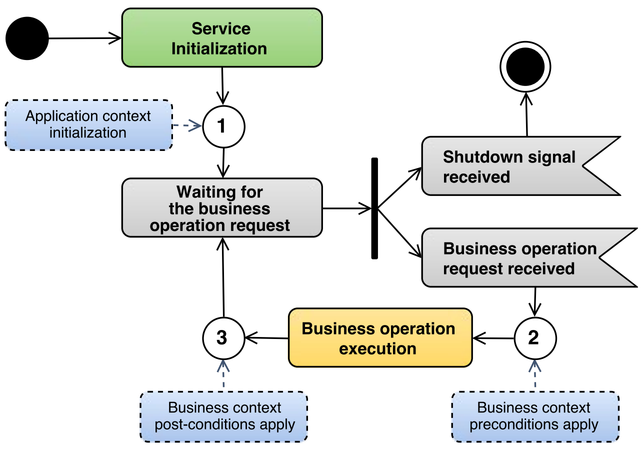
(ii) *Model structure*, or more specifically the structure of the model in the protocol, has to be always considered on both sides of the communication to serialize and deserialize the data. This aspect is usable for verification that both communicating services expect the same protocol structure and there are no inconsistencies.

Second, the *advice* represents the functionality to weave in:

(a) *Business context preconditions* advice is a set of rules to meet before a business operation is executed, e.g., the user is logged in and has the required privileges.

(b) *Business context post-conditions* are rules applied after a business operation is executed, e.g., data filtering based on the logged user's privileges or expected results.

(c) *Business domain configuration* is part of the application context represented by a map of business domain-related


Service Initialization
Application context initialization
1
Waiting for the business operation request
Shutdown signal received
Business operation request received
Business operation execution
2
3
Business context post-conditions apply
Business context preconditions apply
Fig. 2: Service life-cycle and application of the advice

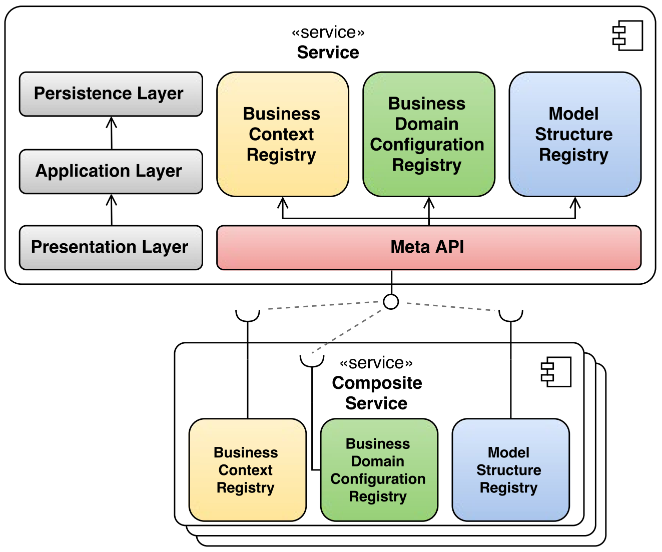


Fig. 3: Service architecture using ADDA for SOA

variables used within a service, i.e., during rules evaluation or business logic execution, e.g., the VAT percentage.

(d) *Model structure* advice contains information about the public business objects defined within each service, i.e., a name of the objects and name and type of each field.

Third, we identify the *join points*, i.e., the points, where advice is applied. Those are denoted in Figure 2:

① First join point triggers during *service initialization* when the service establishes its application context.

② *Before the execution* of a business operation, it validates preconditions of the addressed business context.

③ *After the execution* of a business operation, it applies post-conditions of the business context.

Finally, the *aspect weaving* combines all the advice into proper join points with the respect to current execution context. It is conducted by platform-specific *aspect weavers*, included within each individual service.

B. Architecture

We modify the conventional layered architecture of a service [19] to accommodate the needs of runtime aspect weaving, as displayed in Figure 3. Each service separates the concerns and stores them in registries. This helps to decompose the system into smaller units with a single responsibility. On the other hand, it prevents simple reuse of such information as they are in platform-agnostic form outside the execution point. We apply ADDA to overcome this limitation.

First, the *business contexts* defined by operations of a service, e.g., access policies and order validation rules, are stored in platform-independent DSL in a *Business Context Registry*. Second, the *business domain configuration*, e.g., VAT percentage or business hours definition, is represented by a map of variable names and their values. Those are stored in a *Domain Configuration Registry*. Third, the *Model Structure Repository* maintains the metadata of the structure of its public model. Finally, in order to distribute the information among services, each service must expose its registers through a Meta API. Then other services access this API and retrieve the information they need.

C. Service initialization

When a service starts, it initializes its application context including environmental variables and all business contexts. In an environment with shared business contexts, the service must fetch addressed contexts and business domain configuration from services it depends on. For example, the Order Service requires business contexts and domain configuration defined by the Billing, Shipping, Product, and User services.

First, the Order Service discovers all the other services so that it can contact them. This could be achieved in different ways, depending on particular SOA implementation. For example, we can use Service catalogs or Enterprise Service Bus. For the sake of simplicity, we will not discuss this problem, because it is not relevant to the ADDA approach.

Second, the Order Service downloads the business domain configuration from its dependences. As we have established, the domain configuration is a set of environment variables, so the service merges them into its application context straight away and exposes them in the registry.

Third, the Order Service downloads the business contexts from its dependencies. Then, it compiles them with its own business contexts. Finally, it inserts them into the application context in the platform-specific format, and into the registry in the platform-independent format.

Finally, it extracts the public model structure metadata and stores them within the Model Structure Registry. Then, it verifies the structure of the communication protocol comparing its own metadata to the metadata of the dependencies.

D. Business operation execution

Once the service is initialized and running, it expects requests to execute business operations. For each business operation, there is a business context defining the preconditions and post-conditions.

First, when the execution of a business operation is requested, the aspect weavers intercept the request and check the current business context and validate the applicable preconditions of business rules with the execution context. For example, they verify the user is logged in. If the validation fails, the business operation execution is prevented, and the service returns an error message.

Second, when the execution of the business operation finishes, the aspect weavers intercept the response and apply the corresponding post-conditions to restrict the returned data. For example, they drop a link into the application backend when the user is not an administrator.

E. Summary

Application of ADDA into SOA achieves information reuse while keeps the concerns of each service separated. It maintains the business contexts and business domain configuration in platform-independent format within the individual services and exposes them to other services via API. This allows *easier composition of services*, where one service executes business operations of other services. The service it is able to apply up-to-date restrictions defined by the other services on its input.

This approach reduces overall development and maintenance efforts in the long run. We achieve this through reduction of manual information restatement and separation and reuse of the cross-cutting concerns, i.e., the business rules, the business domain configuration and the model structure. Having them in the single point of truth reduces the size of the codebase, as well as lowers the risk of human error because the restated information does not have to be synchronized manually, which is a highly error-prone activity.

On the other hand, this approach introduces significant initial overhead. It requires implementation of the aspect weavers and registers for each platform. However, these can be reused across services built on the same platform, and also across different projects.

VI. Case Study

In order to evaluate ADDA for SOA and receive a preliminary feedback, we conduct a case study and elaborate how we tackled the challenges discussed in Section III. Consider the e-shop system example introduced in Section II. There are six individual services, which provide different functionality. First, there is the *Users service*, which maintains both customers and employees, and their profiles and privileges. Second, there is the *Store service*, which deals with storage supplies. Third, the *Emailing service* sends e-mails to both customers and employees. Finally, the component *Customer service*, which includes the composite *Order service* maintaining orders, the *Billing service* providing API to the 3rd party billing services, and the *Shipping service* providing a facade to the 3rd party shipping services.

Consider the services are implemented using different technologies, due to the fact that there are multiple teams working on the system. Each team has different a experience and fulfills different non-functional requirements through the solution stack. The *Billing*, *Store*, and *Emailing* services are implemented in Java, because of its reliability and performance. The *Customer* and *Order* services are implemented using Python, because it provides the best libraries for data analysis, and the company needs to analyze the orders to support business decisions. The *Shipping* service is written in server-side JavaScript, because it deals with various third-party APIs and JavaScript provides the most libraries for such tasks.

A. Business rules centralization

First, we implemented the *Business Context Registers* for each platform to persists business contexts and expose them through API. We also tailored a DSL similar to JBoss Drools[4], which a powerful DSL for rule-based systems. We implemented the language in JetBrains MPS[5], which is a tool designed for tailoring custom DSLs. It also provides a parser and a compiler into customizable output. This enabled us to define, store, and distribute business rules in platform-independent format.

Second, we implemented DSL compilers, which merge local and remote[6] business rules and translate them from the platform-independent format into platform-specific executable languages. Moreover, we implemented aspect weavers, which intercept the business operations and apply the business rules advice.

The composite Order service is now able to apply transitive business rules of the User, Billing, and Shipping services without their manual restatement.

B. Configuration centralization

As we stated earlier, the business domain configuration is a special case of service composition. We solved this problem similarly as the business rules centralization. We implemented the *Business Configuration Registers* for each platform. These registers store the configuration variables in a name-value dictionary and provide access to them through API. Then, we also added aspect weavers, which download and merge the business domain configuration from dependencies.

Alternatively, having more powerful DSL, we might declare the configuration as a part of the root business context, i.e., the parent context to all other contexts. Then, we could drop the Business Domain Configuration Registry and all related weavers as the configuration would be included in the Business Context Registry as another context. However, for simplicity, we maintain the configuration separately.

C. Documentation extraction

ADDA approach already provides a mechanism to extract an up-to-date business documentation of a monolithic system [20]. As all the services follow ADDA, we can also extract their documentation. Since each service exposes its metadata through public API, we implemented advanced documentation

[4]https://www.drools.org/

[5]https://www.jetbrains.com/mps/

[6]The business rules definitions are downloaded from Business Contexts Registers of services this service depends on.

generator discovering all services in the system and fetching their metadata. Then, similarly to pure ADDA, we combine the information together to identify the services, their operations and business contexts, their dependencies, and the structure of their public model, i.e., the structure of business objects. The generator implementation follows the suggestion for pure ADDA documentation generator, only it loops over all services and identifies their dependencies.

Having this documentation generator opens new possibilities. First, we can produce the result as HTML to overview the system and archive it or give it to the architects. We can also give it to domain experts to review and validate the flow and business rules. Having the overview of the entire SOA system significantly simplifies their work. Finally, we can produce the documentation in a formal language and then reason over it. For example, we can verify the feasibility of all contexts or find contradictions, which may result from the automated composition of business contexts.

D. Summary

We described the implementation of ADDA concept into SOA to reduce information restatement and simplify development of the system. Furthermore, ADDA for SOA opens new ways to use the extracted and exposed information, e.g., for automated business documentation extraction and its validation and verification. Unfortunately, the efficient implementation requires that all services in SOA follow ADDA for SOA concept. Otherwise, the concern reuse is significantly limited. Next, the concept implementation relies on complex tools as a DSL for business context description and platform-specific aspect weavers, which introduces a significant initial overhead. In consequence, migrating an existing system to ADDA for SOA concept seems to be highly challenging.

VII. Related Work

SOA is one of existing architectural solutions for large applications with difficult maintenance, performance issues, and multiple development teams. Deployment, composition, and maintenance of services belong among the most significant issues. In this paper, we propose a novel approach addressing composition and maintenance difficulties, and this section elaborates them in the context of existing work.

A. The architecture

Nowadays, SOA itself is considered outdated and replaced by a novel and more evolved approaches. Microservices pattern is the leading architecture replacing SOA [9]. However, this architecture preserves existing SOA principles and adds additional constraints addressing deployment and maintenance issues. For example, it emphasizes simple services and rapid delivery. Next, it suggests the use of multiple agile teams and service communication through an independent, usually HTTP-based, protocol such as REST and SOAP. Finally, it stresses decentralization through choreography [8], [21]. Contrary, plain SOA often uses orchestration, e.g., Enterprise Service Bus, which brings centralization.

Nevertheless, the basic principles persist and this work applies to them. The proposal expects distributed environment, independent standalone services, and communication through the network. Development workflow, service deployment or actual composition of services are orthogonal to the approach.

B. Service composition

There are two basic approaches to the service composition. First, the services are orchestrated in a network with a director validating and forwarding messages, or the services know their dependencies and somehow they look up them themselves [8]. While the first more centric approach is known as an *orchestration*, the other is known as a *choreography*. None of these actually apply to the proposed approach. Whether the services are discovered through a service registry such as Universal Description, Discovery, and Integration (UDDI) catalog, their addresses are hard-coded in services, or the configuration is provided by a central component is not significant [22]. The proposed ADDA for SOA approach assumes the existence of the dependencies but does not deal with the implementation of a discovery process.

Novel approaches to service composition often use Artificial Intelligence (AI) due to increasing number of existing services and their complexity [10]. There are these automated composition approaches as manually maintaining and evaluation the services is difficult and exacting. The proposed techniques use AI to optimize deployment of the services into a cloud to utilize the performance, to compose services together, to find the best implementation of the dependency etc. All these are performed based on the conducted analyses by an AI.

Each composition service has to consider at least a subset of the business rules declared by services it depends on, but unfortunately, none of these composition approaches efficiently supports the composition of business rules. There exist too many implementations of service description, discovery, and meta-data extraction techniques that it is nearly impossible to gather and reuse this information. Thus, in this work, we propose the approach focusing on reuse of business rules, which does not interfere with existing service composition approaches.

C. Business rules representation and composition

The major part of this paper deals with extraction, reuse, and composition of business rules within composite services. Inspection of dependencies and extraction of business rules requires suitable and inspectable representation of the rules.

Model-Driven Architecture (MDA) belongs among both major research and industrial approaches to SOA design. It describes the business domain in multiple models on different levels of abstraction to avoid manual information restatement and enable information reuse. The more specific models are generated from the more abstract models using transformation and forward engineering techniques. In the end, the service source code is produced [23]. Unfortunately, this technique suffers from the lack of support of backward transformation, i.e., when the more specific model is modified, we are unable to propagate these modifications into more abstract models. Then regeneration of this model overwrites these modifications. In addition, MDA for SOA usually uses special languages with the better focus on services, service providers, etc., and lacks the support of business rules [24]. Unfortunately, the business rules with their cross-cutting nature are difficult to encapsulate in object-oriented techniques such as MDA [25]. Although there are options such as OCL to extend the models, but they are still unable to encapsulate and reuse repeated rules, they restate them manually instead [18].

Similar intentions as ours are discussed in [26]. The authors claim that business rules are often a subject of change, while implementation of services and SOA structure changes less frequently. Thus, then separation of concerns, more particularly business rules, leads to maintainable implementation. They propose a Business Process Execution Language (BPEL) [27] extension separating the business rules from services and declaring them using DSLs. As business rules are more about declaration what to do than how to do it, they introduce several new central meta-services dealing with business declaration, transformation, and business process interception to trigger actions. These services run rule-based engines to deliver high-performance rules evaluation. While this approach surely simplifies the maintenance, it has significant limitations. First, BPEL is designed for a centric orchestration, while recent research and best practices suggest decentralization through a choreography. Then, having DSLs simplifies maintenance comparing to hard-coding the rules into source code, but their further inspection and transformation is still difficult as there are multiple different languages. Finally, as the rules are part of the orchestration description in BPEL, then when they change, the whole orchestration must be updated. Contrary, our approach is more restrictive about used DSL, but it is agnostic to used composition method. Furthermore, when a single service changes, only the services depending on it are notified by a push event and then are internally reloaded.

The alternative approach focuses on identification of business contexts and reuse of business rules from dependencies [28]. The authors propose a framework for the construction of composite services. For each dependency service, they describe its API including business operations, their preconditions and post-conditions, which is a business context in terms of this paper. Then, using their framework, they produce a composite service considering the contexts of the dependencies. While the intentions are similar, this paper proposes more generic approach. Instead of the manual description of each dependency, it reuses their contexts through inspection of automatically exposed meta-data, which it does through separation of concerns using AOP.

D. Documentation extraction

Maintaining up-to-date business documentation of existing SOA is very challenging. SOA is vast living system and with many performed changes, the documentation gets quickly obsolete. Acquiring then up-to-date documentation is barely possible, we must fall back to reverse engineering methods. Extracting the business documentation, i.e., the list of services, their operations with business contexts and a structure of communication protocol from SOA is basically like extracting it from a monolithic application plus dependencies.

Reverse engineering of monolithic applications is well discussed. For example, we may apply phrasal pattern matching on the source code to extract the rules [29] or construct a call-graph and look for branching [30]. Either way, we must identify the execution context, i.e., variables and their origin used in extracted expressions. Generation of such documentation is challenging and the result may be inaccurate depending on the technology and code conventions. Importance but the difficulty of business rules extraction from legacy information systems is discussed in [31]. The authors propose a semi-automated technique to extract the rules, but as it is obvious, such a documentation would require significant efforts, be inaccurate, and might not be up to date.

The difficulty of business rules encapsulation and subsequent automated extraction lies in their characteristic. As they are considered throughout the whole system, they cross-cut multiple layers, components, and often technologies. Unfortunately, commonly used Object-oriented programming fails in the encapsulation of such cross-cutting behavior [15], and tends to their manual tangling and duplication in a code base. Their separation is very difficult [25] due to the necessity to apply them in various places and technologies [3]. However, there exists an efficient documentation extraction technique for applications using ADDA to separate business rules [20]. Having business contexts described in DSLs and available for transformation, we are able to read this meta-data to construct the documentation. This technique applies to this paper. As we propose, each service uses some implementation of ADDA and exposes this meta-data through public API. Then, we are able to browse the SOA and fetch all meta-data to construct the documentation of the overall system.

VIII. Conclusion

There exist many open challenges in SOA. For example domain decomposition, service discovery, composition, deployment, and evolution, or inter-team communication. In this paper, we focused on the separation of concerns and their reuse among composite services. We proposed a novel aspect-based approach ADDA for SOA. It introduces several new components into a conventional service architecture. They maintain separated concerns such as business contexts describing preconditions and post-conditions of business operations, business domain configuration, and the structure of the public model in the platform-independent format. These isolated concerns are exposed via API to other services. That enables them to fetch this metadata, transform them and combine with their own business contexts and configuration. The model structure is used for the verification of the communication protocol. Besides the simplification of service composition, we show the simplicity of generation of up-to-date business documentation listing the services, their operations, preconditions, post-conditions, and the structure of the communication protocol, which often reflects the structure of business objects.

ADDA for SOA delivers significant maintenance improvement, codebase reduction, and context-awareness to services. It isolates business rules into the single point of truth in DSL and weaves the rules together at runtime with the respect to the current execution context. Use of DSL enables the involvement of domain experts into development. Automated distribution and restatement of business rules remove the need for manual synchronization of all places, which reduces maintenance efforts and lowers the risk of human error.

One the other hand, ADDA itself introduces significant overhead, as it requires design and implementation of the DSL, and platform-specific application-independent aspect weavers. In SOA, there are multiple programming languages and platforms involved, which increases the number of required aspect weavers. Development of the technological stack requires major efforts. Furthermore, all services in SOA have to follow ADDA for SOA concept, otherwise, no automated concerns composition and reuse can happen. Moreover, separation of concerns slightly reduces cohesion and thus maintaining the business rules apart of the related code is more demanding.

There is still work to do in future. Besides the need for the production-ready implementation, we will focus on delivery of larger evaluation of development efforts comparing ADDA for SOA to pure SOA or some its alternative. Finally, we will focus on design and formalization complex but easy to use DSL for business rules in SOA. There is the need for many features such as inheritance, rules modifiers, and declaration of constants. Use of ADDA for SOA also opens new possibilities. For example, having all metadata exposed via API, we are able to maintain business rules for all services from a single place, e.g., a maintenance application. We might visualize the relations, modify the rules, and then let the services update themselves and reload the configuration.

Acknowledgement

This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/234/OHK3/3T/13 and Czech UPE and Avast Foundation grant at the Czech Technical University No. DP17\2017010007.

References

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development*. Pearson Education India, 2012.

[2] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] K. Cemus and T. Cerny, "Aspect-driven design of information systems," in *SOFSEM 2014: Theory and Practice of Computer Science, LNCS 8327*. Springer International Publishing Switzerland, 2014, pp. 174–186. ISBN 978-3-319-04298-5

[4] R. Perrey and M. Lycett, "Service-oriented architecture," in *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 116–119.

[5] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: service-oriented architecture and web services*. IBM Corporation, International Technical Support Organization, 2004.

[6] M. R. Cecil, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[7] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE, 2003, pp. 3–12.

[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *arXiv preprint arXiv:1606.04036*, 2016.

[9] M. Fowler and J. Lewis, "Microservices," *ThoughtWorks. https://martinfowler.com/articles/microservices.html [accessed on March 21, 2017]*, 2014.

[10] J. Rao and X. Su, "A survey of automated web service composition methods," in *International Workshop on Semantic Web Services and Web Process Composition*. Springer, 2004, pp. 43–54.

[11] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[12] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C recommendation*, vol. 26, p. 19, 2007.

[13] M. J. Hadley, "Web application description language (WADL)," 2006.

[14] T. Cerny and M. J. Donahoo, "How to reduce costs of business logic maintenance," in *Computer Science and Automation Engineering (CSAE)*, vol. 1. IEEE, 2011, pp. 77–82.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[16] K. Cemus, F. Klimes, O. Kratochvil, and T. Cerny, "Separation of concerns for distributed cross-platform context-aware user interfaces," *Cluster Computing*, pp. 1–8, 2017.

[17] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[18] K. Cemus, T. Cerny, and M. J. Donahoo, "Automated business rules transformation into a persistence layer," *Procedia Computer Science*, vol. 62, pp. 312–318, 2015.

[19] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 2015, pp. 583–590.

[20] K. Cemus and T. Cerny, "Automated extraction of business documentation in enterprise information systems," *ACM SIGAPP Applied Computing Review*, vol. 16, no. 4, pp. 5–13, 2017.

[21] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016.

[22] E. Al-Masri and Q. H. Mahmoud, "Discovering the best web service," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 1257–1258.

[23] A. Rahmani, V. Rafe, S. Sedighian, and A. Abbaspour, "An mda-based modeling and design of service oriented architecture," *Computational Science–ICCS 2006*, pp. 578–585, 2006.

[24] S. K. Johnson and A. W. Brown, "A model-driven development approach to creating service-oriented solutions," in *International Conference on Service-Oriented Computing*. Springer, 2006, pp. 624–636.

[25] R. Kennard, E. Edmonds, and J. Leaney, "Separation anxiety: stresses of developing a modern day separable user interface," in *Human System Interactions. HSI'09. 2nd Conference on*. IEEE, 2009, pp. 228–235.

[26] F. Rosenberg and S. Dustdar, "Business rules integration in bpel-a service-oriented approach," in *E-Commerce Technology. Seventh IEEE International Conference*. IEEE, 2005, pp. 476–479.

[27] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte *et al.*, "Business process execution language for web services," 2003.

[28] J. I. Fernández Villamor, C. A. Iglesias Fernandez, and M. Garijo Ayestaran, "Microservices: Lightweight service descriptions for rest architectural style," 2010.

[29] E. Putrycz and A. W. Kark, "Connecting legacy code, business rules and documentation," in *Rule Representation, Interchange and Reasoning on the Web*. Springer, 2008, pp. 17–30.

[30] X. Wang, J. Sun, X. Yang, S. Maddineni *et al.*, "Business rules extraction from large legacy systems," in *Software Maintenance and Reengineering*. IEEE, 2004, pp. 249–258.

[31] J. Shao and C. Pound, "Extracting business rules from information systems," *BT Technology Journal*, vol. 17, no. 4, pp. 179–186, 1999.