Abstract

Large systems-of-systems often service large numbers of stakeholders – more stakeholders often means more concerns, many of which are crosscutting. The Rich Services architecture is a type of Service-Oriented Architecture (SOA) that allows hierarchical decomposition of a system architecture into separate concerns, thereby capturing different system aspects and their interactions, and accounting for crosscutting concerns concisely and sensibly. By leveraging emerging Enterprise Service Bus technology, Rich Services enable a simple and direct deployment mapping to a system-of-systems network.

This paper describes a development process that leverages Rich Services and is compatible with agile development methodologies. It encompasses use cases, requirements decomposition, role identification, and service definition to produce a Rich Services logical architecture that can then be mapped to a virtual network topology, and finally a physical network topology. The process decouples development stages to improve flexibility and productivity of complex projects, promising reduced cost and risk.

1. Introduction

One of the key challenges to building systems that meet the needs of large numbers of stakeholders is integrating and reusing existing systems, while adding new functionality. The challenge involves understanding and modeling both the existing systems and the emergent requirements, then creating systems that are robust, performant, and maintainable. Traditional integration approaches involve time-consuming rework bearing major financial and technical risk.

Service-Oriented Architectures (SOAs) have emerged as a widely accepted solution to this challenge; they use standards-based infrastructure to forge large-scale systems out of loosely coupled, interoperable services. SOAs can create systems-of-systems by mapping existing systems into services, then orchestrating communication between the services. New functionality can be created by either adding new services or modifying communication among existing services. Because of these features, SOA projects are particularly amenable to agile development processes. Consequently, well-executed SOAs can drive down financial and technical risk by improving flexibility and reducing time to market.

As the number of stakeholders of a SOA project grows, though, so typically does the number and complexity of various business concerns (e.g., governance, security, and policy) and their level of distribution across the architecture. In order to maintain SOA’s advantages, the integration of these concerns requires a framework that is scalable to complex systems and provides decoupling between the various concerns. Our Rich Services architecture is a type of SOA that accomplishes this while providing direct and easy deployment mapping to runtime systems such as Enterprise Service Buses (ESBs).

In this paper, we show how the Rich Services architecture can be leveraged into an end-to-end software engineering process that ranges from use case elicitation to physical network deployment. We describe how the hierarchical framework manages problem complexity by decomposing complex problems into primary and crosscutting concerns, providing flexible encapsulation for these concerns, and generating a model that can be easily leveraged into a deployment. The process is geared, specifically, towards rapid systems-of-systems integration.

The remainder of this paper is structured as follows. In Section 2, we present Rich Services both at the logical and deployment levels. In Section 3, we introduce a development process that guides the applicability of Rich Services in complex systems with a variety of crosscutting concerns. In Section 4, we discuss lessons learned from our case studies and present related work. Conclusions round out the paper.

2. Architectural blueprint

In this section, we present our architectural vision of Figure 1 for addressing the complexity of various business concerns in large-scale systems. In the following, we explore this architectural blueprint both as a logical model and as a guide to a deployment architecture. The direct mapping between the logical architecture and its implementation simplifies the evolution of the system and contributes to the practicality of the approach.
2.1. Logical view

Figure 1 depicts our logical service-oriented architecture, inspired by ESB architecture/implementations [8][10][11]. The architecture is organized around a message-based communication infrastructure and is hierarchically decomposed into Rich Services, which encapsulate various capabilities/functionalities. The interaction between a Rich Service and its clients is accomplished via a Service/Data Connector. To manage the service orchestration, the communication infrastructure has two main layers. The Messenger layer is responsible for transmitting messages between services. The Router/Interceptor layer is responsible for intercepting messages placed on the Messenger and then routing them among all services involved in providing a particular capability.

The main entity of the architecture is the notion of Rich Service. We distinguish between Rich Application Services (RASs) and Rich Infrastructure Services (RISs). RASs provide core application functionality, mandating the business flow. RISs do not initiate any communication by themselves, but reroute or filter messages defined by RASs. Examples of RISs are policy enforcement, encryption, and authentication. New services can be plugged into the architecture without changing the existing services. To integrate an encryption mechanism, for instance, only the communication infrastructure needs to be aware of the encryption RIS: the Router/Interceptor changes the routing tables to ensure that every message sent to the external network is first processed by the encryption service.

Rich Services are connected to the communication infrastructure via Service/Data Connectors. A Rich Service could be a simple functionality block or it could be hierarchically decomposed into further Rich Services. The Service/Data Connector encapsulates the internal structure of a service and exports only the interfaces that the connected Rich Service provides externally. We associate both a structural and a behavioral view with the interfaces.

The communication infrastructure enables loose coupling and seamless communication between services. The use of a Router/Interceptor layer removes dependencies between services and their relative locations in the logical hierarchy. Thus, services from different levels of the hierarchy - possibly from different authority domains - can interact with each other seamlessly with the help of appropriate infrastructure services and routing tables.

2.2. Deployment view

The architecture in Figure 1 can also be used as an implementation architecture leading to an ESB solution. ESBs provide message routing amongst well-defined, loosely coupled, coarsely granular services similar to our notion of Rich Services. They achieve cost efficiency and flexibility through the integration and reuse of services. In addition, ESBs may employ any of several routing strategies (e.g., static routing, content-based rules, and itinerary) to achieve architectural goals such as distributed execution, load balancing, flow throttling, auditing, redundancy, failover, online maintenance, provisioning, and minimal service interdependence. This makes them particularly attractive as deployment architectures.
Thus, the instantiation of the architecture leverages existing technologies such as the Mule ESB framework [11]. In this case, the Mule transport layer corresponding to the Messenger element of the logical architecture can be leveraged to connect both components local to Mule implementing some functionalities and generic external components. External components are connected to the Mule framework using some of the supported transport technologies. For instance, they could be implemented as Web services and connected through the Mule SOAP transport provider. Furthermore, external services can be another implementation of Rich Services.

The close alignment of the logical architecture for the Rich Services and the ESB deployment architecture yields a direct logical-to-deployment concept mapping, though other deployment architectures would also work.

3. Development process

Traditional component-oriented software development methodologies regard components as the building blocks of an architecture. However, system requirements are generally not defined in terms of architectural components. Instead, they typically span across the various components of the system, establishing complex interaction dependencies. Component-centric development favors simplicity of structural decomposition over orchestration simplicity. The price to pay lies in increased integration challenges, where the crosscutting orchestration concerns have to be reconciled.

In this section, we introduce a service-oriented development process that achieves a clean separation of the logical model of the system and its implementation. This process fits well with large-scale system-of-systems integration. To model a system using our Rich Services architecture, we employ an iterative development process with three phases, each with one or more stages. Each phase (and also the entire sequence of phases) can be iterated several times to refine the system architecture and obtain an implementation that satisfies the requirements of the system. This is a key feature because crosscutting concerns can be abstracted later in the development process.

The first phase, Service Elicitation, captures the system requirements in a service repository and a corresponding role domain model integrated with crosscutting system concerns such as security, access control, confidentiality, encryption, error handling, tracing, and transaction support. For each concern, we can leverage an existing technique of requirements gathering. For example, for security we can employ elements from the Common Criteria [16] to determine assets, risks, and mitigation strategies. In the automotive domain, the approach of SPUR [15] can be used for modeling increasingly important attributes such as security, privacy, usability, and reliability.

We begin by capturing into use cases the general functional requirements of the system, together with additional stakeholder concerns, such as governance, security, and fault tolerance (see Figure 2). The resulting use-case graph also stores the dependencies (e.g., inclusion) between different use cases.

We analyze the use cases and construct a domain model of the system with all logical entities, data structures, or other knowledge representations. Note that we perform requirements gathering in all stages of the development process. Thus, we can operate with partial specifications for crosscutting concerns that are identified at the use-case stage and are input to the other stages, where we detail them and identify new concerns.

From the system domain model, we identify the roles as unique, abstract entities interacting in each use case. Roles are logical representations of collaborators in a service orchestration. Examples are the client and server roles in a client/server or peer-to-peer network. Depending on the network type, these roles are played by some or all of the network nodes in a concrete deployment.

We define services as interaction patterns between roles [6] for the realization of each use case. Each service “orchestrates” interactions among system entities to achieve a specific goal [4]. Within a service, roles exchange messages, thereby switching from one state to another. A Message Sequence Chart (MSC) [5] can be used to capture the interaction and various role states. The complete set of states and state transitions of a role obtained from all services it participates in captures its full behavior.

A set of crosscutting concerns may apply to the interaction pattern between roles within a service. The complete set of services defined at this stage constitutes the service repository of the system. Next, we obtain the role domain model through a consolidation of the dependency knowledge between roles participating in the identified services.

The second phase, Rich Services Architecture, defines a hierarchic set of Rich Services as a logical model of the system out of the role domain model and the service repository. Based on client values such as architecture comprehensibility, performance, and organization domains, we establish the hierarchy of the RAS from the role domain model. There are two ways for mapping the identified services and their roles to the Rich Service notion.

The first option is to map each role to a corresponding RAS, then encapsulate the services in which it participates into the routing mechanism of the Rich Services architecture. Through the Service/Data Connector of the resulting Rich Service, we expose to the upper layers the service itself and allow it to act as a higher-level role.

The second option is to directly map each service to a RAS that can be basic or composite. This is useful when the roles have an actual correspondence to existing system
infrastructure (e.g., database) that can readily be used. In this case, we obtain a simplified view of the first option by ignoring the lowest levels of the architecture (the leaves of the tree hierarchy).

The crosscutting concerns map to RISs that can also be decomposed and may even have internal Infrastructure Services of their own. Crosscutting concerns include quality requirements, which may be identified in the stage of defining the services. MSCs can be augmented with these requirements [7]. For instance, for run-time monitoring of QoS properties, we can add a QoS Monitor as a separate infrastructure service connected to the Router/Interceptor layer. As we map MSCs to the roles behavior, we can also add an Interaction Monitor as a RIS. At run-time, each message will be routed through these monitors, offering the possibility of assessing the correctness and the quality of the services. Because the logic needed to orchestrate the message flow is captured by an MSC, we can leverage our work on state machine generation to synthesize it [6]; another option is to describe the orchestration logic by means of WS-BPEL.

Each Rich Service in the hierarchy presents itself to its environment via a dedicated Service/Data Connector that hides its internals. In the process of mapping roles and services to Rich Services, we set up the communication channels, decide the location of each concern, set the routing tables, and decide what to publish through the connectors. In particular, we associate with every Rich-Service interface the MSCs that define the communication patterns/protocol between roles outside the Rich Service (Imported Roles) and roles inside it (Exported Roles).

The collection of Rich Services defined at this point represents the Rich Services logical model of the system, or Rich Services Repository. Each of these steps feeds back into the definition of the use cases; hence, we can refine the model without worrying about the deployment architecture. The newly identified use cases translate at the next iteration into individual Rich Services, and the relationships between the use cases translate into corresponding relationships between Rich Services.

The third phase, System Architecture Definition, establishes a relationship between the Rich Services model of the system and its implementation. Because the target of our approach is large-scale system-of-systems integration, we first analyze the topology of the system in terms of computational and storage nodes and available networks. When such systems are not yet available, the logical model from the second phase can provide hints for building and configuring an adequate system topology.

In the second stage, we create an idealized network of the identified Rich Services, where each RAS is represented as a virtual host connected to a common bus. This stage focuses on logical connections between RASs, recognizing that duplication of services may occur.

Afterward, we perform a mapping of the Rich Services virtual network to the middleware running on the physical
hosts of the systems. Several optimizations for performance, security, and other crosscutting concerns are possible at this stage. As an example, the logical Rich Service architecture can be mapped exactly 1:1 to an ESB deployment architecture. Depending on the system requirements and the available resources, some services can be duplicated, whereas other duplicated services can be replaced with proxies for unique services. Furthermore, some levels in the hierarchy can be flattened.

In the final stage of the development process, we proceed to the actual implementation of both RAS and RIS. Depending on the middleware infrastructure, COTS software components can be used to implement the functionalities of the Rich Services. Partial system specifications may be used to generate prototype implementations.

In addition, when system requirements cannot be accomplished by the current architecture, the implementation can be used to refine the Rich Services architecture and even to provide additional use cases and concerns in a new cycle of the development process.

4. Discussion and related work

We have encountered examples of concerns integration within a wide range of interdisciplinary projects conducted at the California Institute for Telecommunications and Information Technology (Caliit2), with dozens to hundreds of stakeholders involved. Examples include the BioNet, RESCUE [14], and ORION-Conceptual Architecture [13] projects. BioNet seeks to coordinate military and civilian capabilities to detect, characterize, and manage the consequences of a biological attack. RESCUE targets the gathering of information within emergency response networks and the dissemination of it to the general public. ORION-CA supports scientific discovery in global ocean observatories by providing eligible oceanographers ubiquitous access to instrument networks for sensing and actuation, computational resources, and modeling and simulation facilities.

All of these projects have requirements for integrating heterogeneous trust, security, and privacy zones. Each stakeholder brings its own business processes, capabilities, and requirements. We learned that these concerns should be treated up front in any project and refined through all phases of the development process. Rich Services enable hierarchical structuring of the stakeholders’ logical roles and encapsulation of crosscutting concerns according to their individual policies. An iterative process is an effective approach in complex systems as it supports rapid prototyping and can address changing requirements. We also used ESBs in enterprise integration efforts with good results [8], as they provide a flexible and adaptable infrastructure and allow us to remove the coupling between services and the transport medium. Furthermore, the Rich Services architecture eases writing DoDAF [12] style architecture documents as it gives a short distance (a simple mapping) between operational and systems views.

We used the blueprint architecture to also ease the integration of COTS components by lifting the integration concern at the service specification level and by treating it as a first-class citizen in the development process [10]. We explicitly modeled the interaction between the roles defined in the services and the COTS components supplying the functionality. Infrastructure services are plugged into the architecture to execute the data transformations and other glue code for adaptations. This approach reduces the costs and risks of reusing COTS components.

The architectural blueprint leverages well-known enterprise integration patterns and the asynchronous messaging pattern. The communication is carried out by Messages exchanged over Messages Channels [2]. To remove service interdependencies, the Router/Interceptor is an instance of the Mediator pattern [1] and the architecture uses the Plugin pattern [3]. Furthermore, the Façade, Proxy, and Adapter patterns [1], as well as the Messaging Gateway pattern [2] inspired our Service/Data Connector. Finally, the Composite pattern [1] enables the hierarchical nesting of Rich Services.

There are a number of research threads in areas related and tangential to SOA-oriented process engineering: development processes for ESBs, aspect-oriented programming in distributed computing environments, and mapping virtual networks to physical networks.

ESB development process: Most ESB development process literature describes patterns that can be implemented using an ESB, and discusses specific implementation technologies and philosophies. [17] assumes a preexisting collection of services, and shows how ESBs can be used to implement a number of standard patterns. [18] assumes preexisting services, and presents copious pattern based and case-study based implementation guidelines. [2] discusses basic messaging technologies, and presents philosophies of usage and composition leading to real applications. [19] presents the ODSI architecture, a service integration model identifying workflow, and coordinating services to achieve the flow.

The Rich Services approach presents an end-to-end architecture model and process, which complements and incorporates the programming/design approach described in [17][18][2]. These approaches can advise which existing services may be used, the particular form of the MSCs, and the decisions made in the combination and deployment stages. Because the Rich Services process is iterative, it makes good use of the refactoring capabilities inherent in the extensive pattern orientation of the programming/design approaches; this contributes to design and implementation stability.

The Rich Service process parallels the ODSI approach,
but provides guidance on role identification, concerns, services, and the interactions between them. It also models fractally complex relationships. The ODSI approach focuses on workflow description and implementation.

**AOP and distributed computing:** Under Rich Services, Aspect-oriented Programming (AOP) techniques can freely be used for services defined within limited computing domains [9] (e.g., individual processes on individual machines). While AOP can be defined to execute across a network (via RMI calls embedded within aspect code), the result can be difficult to read, maintain, and configure. [20] provides a means for implementing crosscutting concerns across a network.

By focusing on message exchanges, Rich Services simplifies crosscut modeling, and provides a deployment framework that is integrated with the overall service deployment. The Rich Services approach simplifies loose coupling between services, and achieves easier maintainability across design and implementation iterations.

**Mapping virtual to physical Networks:** [21] describes execution of SOA applications on a virtual network that is late-bound to a physical network, essentially creating an SOA overlay network. This scheme can be incorporated into a Rich Services process, thereby shifting the virtual network mapping to runtime, and taking advantage of a network manager’s ability to dynamically allocate services to available network resources.

## 5. Conclusions

The Rich Services process is an end-to-end process that leverages use cases and the Rich Services architectural pattern to produce an efficient and effective ESB physical network deployment. It supports agile methodologies by allowing iteration between phases and decoupling phases. It complements existing ESB development methodologies, is compatible with existing AOP methods, and can leverage virtual networking strategies. It has been demonstrated at Calit2 on three enterprise-scale projects. As such, it is a promising technique for architecting and maintaining complex system-of-systems solutions.

### Acknowledgements

Our work was partially supported by the NSF within the projects “RESCUE” (award #0331690) and “Responsephere” (award #0403433), as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2).

### References


