Unifying Services and Resources: A Unified Architectural Style

José C. Delgado
Instituto Superior Técnico, Universidade de Lisboa, Portugal

ABSTRACT

Current integration solutions are still based on technologies developed for the original Web problem, which is, browsing remote hypermedia documents with text as the main media type. Text-based data description (e.g., XML, JSON) and a stateless and connectionless protocol (e.g. HTTP) are still the norm to achieve distributed integration. However, the Web today is much more dynamic, in which resources are no longer passive hypermedia documents but are active and implement services. SOA and REST are the most used architectural styles to implement distributed integration and each exhibits advantages and disadvantages. This chapter illustrates that they are dual architectural styles; one oriented towards behavior and the other towards state, and contends that it is possible to combine them to maximize the advantages and to minimize the disadvantages. A new architectural style, designated Structural Services, is proposed and described. Unlike REST, resources are able to offer a variable set of operations and, unlike SOA, services are allowed to have structure. To minimize resource coupling, this style uses structural interoperability, based on the concepts of structural compliance and conformance, instead of schema sharing (as in XML) or standardized or previously agreed upon media types (as in JSON). To delineate how this style can be implemented, a new distributed programming language is presented.

INTRODUCTION

The main integration technologies available today are based on the SOA (Erl, 2005) and REST (Webber, Parastatidis & Robinson, 2010) architectural styles, with Web Services (WS) and REST on HTTP as the main workhorses. They constitute two different approaches to the same problem: application integration. SOA has the goal of a low semantic gap, since it models real world entities by resources with services that express their capabilities. This is good in modeling terms, but entails a coupling between the provider and the consumer that hampers dynamic changeability and adaptability. If the provider's interface changes, the consumer's needs to change in accordance. There is no apparent structure, since service composition is hidden behind each service interface.

One of the main goals of REST is to reduce the coupling between provider and consumer, both to increase scalability and adaptability. Real world entities are modeled in a data-oriented way by resources, all with the same syntactical interface (same set of operations). Semantics are restricted to a set of data types (or schemas), either standardized or previously agreed upon between the interacting entities. The variability of the characteristics of entities is modeled by visible structure (resources composed of other resources) and the semantics of the agreed data types.

Unfortunately, the decoupling goal of REST is somewhat elusive. Messages cannot be understood simply by exploring their data structure, touching links blindly. Semantics and behavior need to be considered as well, and this is determined by the type of resources used. If, for example, the provider decides to change its specifications, the code at the consumer will most likely be unable to cope with that.

What happens, in practice, is that REST on HTTP is simpler to use than SOA and many applications are simple enough to adopt a data-oriented interface, REST style. This means that, although REST represents a modeling shift from real world entities (lowering the modeling level and increasing the semantic gap), it is still simpler to use than a full-blown SOA environment. Unless, of course, the application is sufficiently complex to make the semantic gap visible and relevant enough. This is why REST is preferable in simpler applications and SOA is a better match for complex, enterprise-level applications.

Nevertheless, REST gets one aspect right: in a distributed context, interoperability has to be based on structural composition of previously known entities. Forcing these to have one single set of operations, however, does not increase adaptability in the general sense; it only leads applications to adopt a data-oriented style, which is not adequate for all classes of applications.

The following questions then arise:

- Why do we have to choose between the service-oriented (SOA) and data-oriented (REST) styles, instead of combining both and using the best approach for each part of the application?
- How can we increase adaptability without enforcing some particular application style?

This chapter revisits the integration problem with an open mind, without being restricted a priori by existing technologies. The only assumption is that there are entities that need to interact, by using messages. Then, an integration model is defined and its various characteristics compared with those of current technologies, showing how this model can solve some of their limitations.

The main goal of this chapter is to propose and describe a new architectural style, designated Structural Services, which combines the best characteristics of the SOA and REST styles. On the one hand, services have a user defined interface which can be published, discovered and used. On the other hand, services are implemented by resources, which have structure (composed of other resources). Operations are first class resources and messages are themselves resources, able to include references to other resources. As a result, applications can be designed in SOA or REST styles, according to the needs of the application.

This approach does not entail new integration concepts, but rather a new way to combine those that already exist in SOA and REST. Although these styles have been compared (Adamczyk, Smith, Johnson, & Hafiz, 2011; Castillo, et al., 2013) and attempts to combine both styles in one application have been described (Muracevic & Kurtagic, 2009; Li & Svard, 2010), the theme is usually brought up as a decision on which style is the most adequate (Pautasso, Zimmermann, & Leymann, 2008)) or as a transformation of one style to the other (Peng, Ma, & Lee, 2009; Erl, Balasubramanians, Pautasso, & Carlyle, 2011; Upadhyaya, Zou, Xiao, Ng, & Lau, 2011). This is a direct consequence of the lack of an architectural style and of an integration technology that readily supports both SOA and REST. This chapter attempts to take a first step in that direction.

This chapter is structured as follows. We start by describing the integration problem, as well as motivation scenarios. Next, we analyze current integration technologies and derive a hierarchy of architectural styles, justifying the conception of the Structural Services architectural style. We establish a generic model of resources and their services, with decoupling achieved by partial interoperability, based on the concepts of structural compliance and conformance. After a motivating example that exposes the duality and limitations of SOA and REST, we describe the main characteristics of the Structural Services style and how it can be implemented, with a language with native support for it. We also discuss the role of this architectural style and the rationale underlying it. Finally, we provide hints on future research directions and draw some conclusions.

BACKGROUND

Interoperability, as a means to achieve integration, is as old as networking. Whenever two or more resources need to interact, an interoperability problem arises. This has been studied in the most varied domains, such as enterprise collaboration (Jardim-Goncalves, Agostinho, & Steiger-Garcao, 2012), e-government services (Gottschalk & Solli-Sæther, 2008), military operations (Wyatt, Griendling, & Mavris, 2012), Cloud computing (Loutas, Kamateri, Bosi, & Tarabanis, 2011), healthcare applications (Weber-Jahnke, Peyton, & Topaloglou, 2012), digital libraries (El Raheb et al., 2011) and metadata (Haslhofer & Klas, 2010).

Service interoperability (Athanasopoulos, Tsalgatidou, & Pantazoglou, 2006) implies that interaction occurs according to the assumptions and expectations of the involved services. This involves several levels (Mykkänen & Tuomainen, 2008), such as communication protocol, message structure, data format, service syntax, semantics and composition (Khadka et al., 2011) and even non-functional and social aspects (Loutas, Peristeras, & Tarabanis, 2011). This is an area of active research, with interoperability levels above interface syntax still largely dependent on manual or semi-automated work from developers and architects.

Several frameworks have been conceived to systematize interoperability and integration, such as Athena (Berre et al., 2007), LCIM (Wang, Tolk, & Wang, 2009), the European Interoperability Framework (EIF, 2010) and the Framework for Enterprise Interoperability (FEI) (Chen, 2006), which served as the basis of the CEN EN/ISO 11354-1 standard (ISO, 2011). The European project ENSEMBLE embodies an effort to formulate a science base for enterprise interoperability (Jardim-Goncalves et al., 2013).

Existing interoperability frameworks tend to build on existing integration technologies, such as XML, Web Services and RESTful APIs, focusing on how to integrate complex systems by using them. Therefore, they focus more on mimicking the current stage of development of these technologies than on modeling the interoperability problem in its true dimensions, independently from specific technologies. However, these mechanisms are based on HTTP, a protocol conceived for the original Web problem, which is, browsing remote hypermedia documents with text as the main media type (Berners-Lee, 1999).

The main goal was information retrieval, from humans to humans, with client-server dichotomy and scalability as the main tenets. This goal justified many decisions taken then, that given the explosive success of the Web, became standard and are still conditioning current technologies and imposing relevant restrictions on the general communication model required by service-oriented systems (Reuther & Henrici, 2008). In particular, it does not support full-duplex long running sessions required by general services such as those found at the enterprise level. This has spurred workaround technologies such as AJAX (Holdener, 2008) and Comet (Crane & McCarthy, 2008). WebSockets, now part of the HTML5 world (Lubbers, Albers, & Salim, 2010), remove this restriction, add binary support and increase performance.

Even at the human level, the current Web is very different from the original Web. Instead of a static Web of documents, we now have a Web of services, or at least a Web of dynamic documents, generated on the fly from databases or under control of server-side applications. The classic client-server paradigm is also taking a turn with one of the latest developments, real-time communications in the Web (Loreto & Romano, 2012) directly between browsers, by using WebRTC (Johnston, Yoakum, & Singh, 2013) in a peer-to-peer architecture. Nevertheless, this is being done in an evolutionary way, with essentially JavaScript and WebSockets (Lubbers, Albers, & Salim, 2010), to circumvent some of the limitations of HTTP. The most important point to note is that finally humans are also becoming direct Web producers and not merely clients of servers.

The Internet of Things (Gubbi, Buyya, Marusic, & Palaniswami, 2013) is embodying a new revolution. There are now many more computer-based devices than people connected to the Internet and the ratio will

increase in the next few years, according to estimates from the European Commission (Sundmaeker, Guillemin, Friess, &Woelffle, 2010). The consequence of this is that messages in the Internet are now dominated by machine-to-machine interactions, rather than between human users and servers. The mismatch between current technologies and the needs of the new Internet usage context is increasing, since the interaction between machines and between people differ in many ways.

Data interoperability is still largely dominated by text (e.g., XML and JSON). The ability to self-describe data with a schema is one of XML's strongest points, albeit a great source of complexity. JSON is a simpler data format, but very popular in browser-based applications given that it is based on a subset of JavaScript and is a natural format for client-side processing. There is now a proposal for a data schema (Galiegue & Zyp, 2013).

In terms of behavior interoperability, the world is now divided into two main camps: service (Erl, 2008) and resource (Webber, Parastatidis, & Robinson, 2010) oriented architectural styles (SOA and REST, respectively).

SOA, most commonly implemented by Web Services, emphasizes behavior (albeit limited to interfaces, with state and structure hidden in the implementation). REST follows the principles defined by Fielding (2000) and emphasizes structure and state, by exposing inner URIs (with interaction and application state separated and stored in the client and server, respectively, and behavior hidden in the dynamically changing structure and in the implementation of individual resources). Web Services are technologically more complex, but their model is a closer match to real world resources. REST is simpler and finer grained, but leans towards some restrictions (such as interaction statelessness) and is lower level (higher semantic gap between application concepts and REST resources), which for general business-like applications means more effort to model, develop and maintain.

SOA and REST are not competitors, but complementary approaches, each naturally a better fit to different areas of application domains (Pautasso, Zimmermann, & Leymann, 2008). What is lacking is a way to bridge them and to tune up more to one side or another, according to the needs of a particular application.

Another important issue is that neither SOA nor REST directly support behavior implementation, which must be supplied by components implemented in a generic programming language (such as Java or C#) or by BPEL processes, which can be used both with SOA (Henkel, Zdravkovic, & Johannesson, 2004) and REST (Pautasso, 2009). In any case, another technology is needed. To support the service paradigm in an integrated way, behavior elements (instructions and/or operations) should be integrated with the data elements.

We are also interested in compliance and conformance as the foundation mechanisms to ensure partial interoperability and thus minimize resource coupling. These mechanisms have been studied in specific contexts, such as choreography (Bravetti & Zavattaro, 2009; Diaz & Rodriguez, 2009), modeling (Kim & Shen, 2007; Kokash & Arbab, 2009), programming (Läufer, Baumgartner, & Russo, 2000; Adriansyah, van Dongen, & van der Aalst, 2010) and standards (Graydon et al., 2012).

The integration problem

Integration (Mooij & Voorhoeve, 2013) can be defined as the act of instantiating a given method to design or to adapt two or more systems, so that they cooperate and accomplish one or more common goals. What these words really mean depends largely on the domain which the systems belong to, although there is a pervasive, underlying notion that these systems are active and reacting upon stimuli sent by others, in order to accomplish higher level goals than those achievable by each single system.

To interact, systems must be interoperable, i.e., able to meaningfully operate together. Interoperability (Wang, Tolk, & Wang, 2009) is a characteristic that relates systems with this ability and is defined by the 24765 standard (ISO/IEC/IEEE, 2010) as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged". This means that merely exchanging information is not enough. Interacting systems must also be able to understand it and to react according to each other's expectations.

Interoperability is distinct from integration. Interoperability is a necessary but not sufficient condition for integration, which must realize the potential provided by interoperability. This is usually done by resorting to a model, a framework, a method and an underlying architecture, which can be defined as:

- A *framework* is a set of principles, assumptions, rules and guidelines to analyze, to structure and to classify the concepts and concerns in a given domain.

- A *model* is an abridged representation of a given set of systems and their relationships, in the form of a simplified specification of that set (abstract view of the problem, stemming from analysis).

- An *architecture* is a specific organization of a set of systems, providing a concretization of a model in the form of a simplified implementation of that set (abstract view of the solution, stemming from design).

- A *method* is a set of techniques, best practices and guidelines to yield a solution to a problem (in a given set of systems), by building a model and deriving an architecture, in the context of a framework.

In this chapter, we concentrate on the first three, in a domain of computer-based resources and in a distributed and heterogeneous context. Figure 1 illustrates a complex scenario, with several systems, networks and applications that require integration.

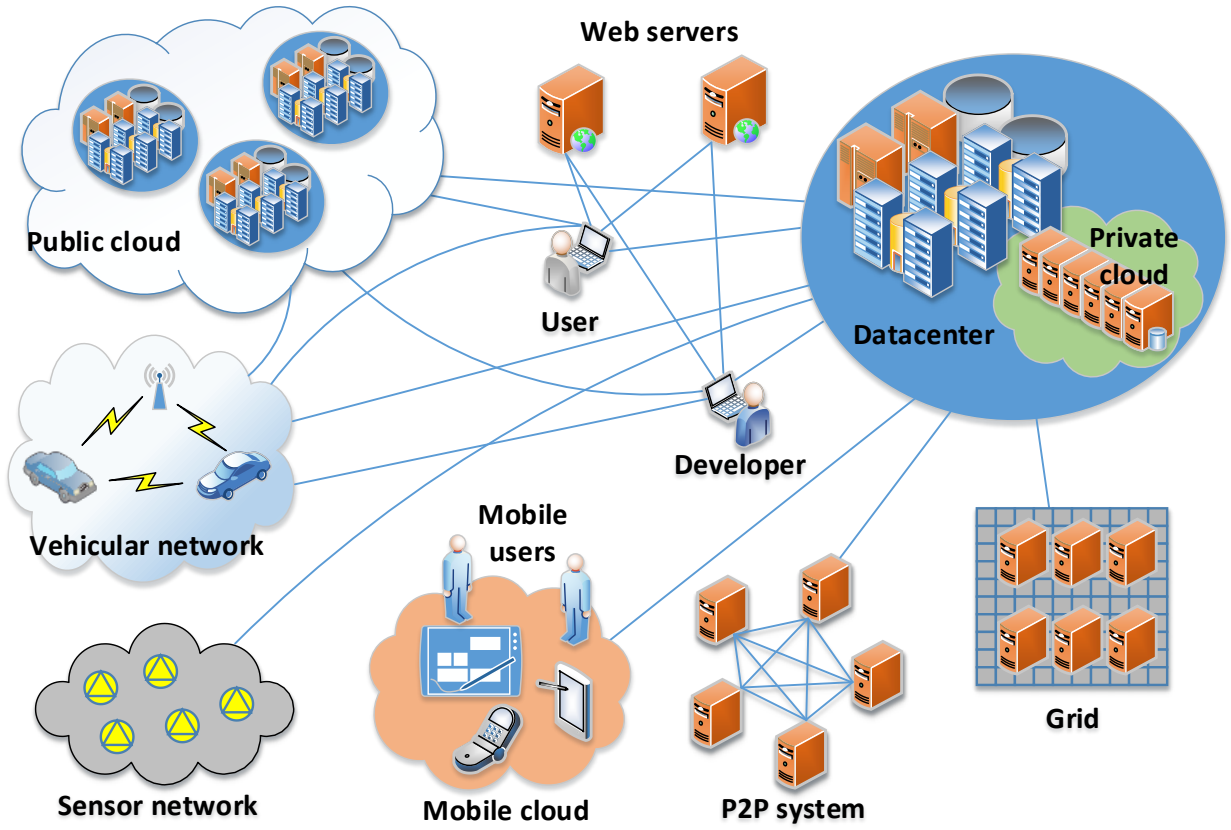


Figure 1. *Scenarios of system and application integration*

Gone are the days when integration was essentially limited to enterprise information systems, located in specific datacenters. Today, Cloud computing (Armbrust et al., 2010) and the Internet of Things (Gubbi, Buyya, Marusic, & Palaniswami, 2013) are revolutionizing the society, both at the enterprise and personal levels, in particular in urban environments (Schaffers et al., 2011) with new services and applications.

In Figure 1, integration will be needed in situations such as:

- Enterprise information systems need to cooperate, reflecting enterprise value chains. Enterprise datacenters still exist, to hold critical or sensitive data, but they it will probably use a private Cloud instead of a mere cluster.

- Enterprise applications will be deployed to hybrid Clouds, integrating the enterprise's owned infrastructure with one or more public Clouds. Integration between enterprises needs to take this into account.

- Mobile Cloud computing (Fernando, Loke, & Rahayu, 2013) is another possibility, given the ever increasing pervasiveness of smartphones and tablets that created a surge in the BYOD (Bring Your Own Device) tendency (Keyes, 2013)

- Grids (Liu, Rong, Jun, & Ping, 2011) and P2P systems (Hughes, Coulson, & Walkerdine, 2010) can be used for batch processing and specific applications, raising specific integration needs.

- The explosive development of the IoT (Internet of Things) (Gubbi, Buyya, Marusic, & Palaniswami, 2013) and of RFID tags (Aggarwal & Han, 2013)) in supply chains raises the need to integrate the enterprise applications with the physical world, including sensor networks (Potdar, Sharif, & Chang, 2009) and vehicular (Hartenstein & Laberteaux, 2008) networks. The European Commission (Sundmaeker, Guillemin, Friess, & Woelffle, 2010) estimate that by 2020 the number of Internet-capable devices will be on the order or 50-100 billion devices, with a ratio of mobile machine sessions over mobile person sessions on the order of 30. This means that the Internet is no longer dominated by human users, but rather by smart devices that are small computers and require technologies adequate to them, rather than to full-fledged servers.

For simplicity, not all possible connections are depicted in Figure 1, but the inherent complexity of integrating all these systems is easy to grasp. Such heterogeneous scenarios have been compared to a jungle of computer-based systems, by using the designation jungle computing (Seinstra et al., 2011). We prefer to think that the Web is evolving into a Mesh, to emphasize heterogeneity, dynamicity and unreliability; characteristics already fundamental to mesh networks (Bruno, Conti, & Gregori, 2005). In contrast, the Web is rather homogeneous (in architecture and protocols), static (DNS and server topology change slowly) and reliable (thanks to the Internet backbone).

The world is increasingly dependent on computers, generating and exchanging more and more data, either at business, personal or event levels. A growing tendency is to analyze big sets of data (Big Data), trying to mine information that can be relevant in specific contexts, such as Business Intelligence and services extracting emergent behavior from a large number of events. Zikopoulos et al. (2012) characterize Big Data by three main properties, referred to as the three Vs:

- Volume (data size and number)
- Velocity (the rate at which data are generated or need to be processed)
- Variety (heterogeneity in content and form)

Essentially, "*Big*" means too complex, too much and too many to apply conventional techniques, technologies and systems, since their capabilities are not enough to handle such extraordinary requirements. This raises the integration problem to a completely new level, in which conventional integration technologies (such as HTTP, XML, JSON, Web Services and RESTful APIs) expose their limitations. These technologies were conceived initially for human interaction, with text as the main format and subsecond time scales, not for heavy-duty, machine-level binary data exchange. New solutions are needed to deal with the new integration problems.

Architectural Styles for System Integration

The main architectural styles (Dillon, Wu, & Chang, 2007) used today in distributed systems are SOA (Erl, 2005) and REST (Webber, Parastatidis, & Robinson, 2010). An architectural style can be defined by one of two approaches:

- Bottom-up, as a collection of design patterns, guidelines and best practices.
- Top-down, as a set of constraints on the concepts and on their relationships (Fielding & Taylor, 2002).

We follow the latter because a rationale should be established before recognizing design patterns and elaborating on best practices. Therefore, we describe an architectural style by constraints with respect to another, less detailed, instead of enumerating its characteristics. This allows us to establish a hierarchy of

architectural styles, as illustrated by Figure 2. These are just the most relevant to this chapter. Others could be identified in specific domains, such as Enterprise Architecture.

At the top, the Null style (Fielding & Taylor, 2002) corresponds to not imposing any constraints at all, not even the existence of resources or any of the other entities depicted in Figure 2. The other architectural styles refine and specialize progressively the Null style by including additional constraints, while reducing the spectrum of entities to which they can apply. For example, the System style assumes resources of any kind, including software applications, people, organizations and other non-computer based entities, but the Software style restrains resources to software modules (used to virtualize non-software entities, such as browsers representing human users).

This style is further specialized by the Object-Oriented and Distributed styles, which in turn are the basis for the most used architectural styles at the Web level, SOA (Erl, 2005) and REST (Webber, Parastatidis, & Robinson, 2010). The Process style, with workflows that invoke services, is also rather popular, in particular in business applications (Marjanovic, 2005). One style can be a specialization of more than one style, as illustrated by the SOA style, which combines both object-oriented and distribution constraints, and the Structural Services style, proposed in this chapter, which specializes in both the SOA and REST styles.

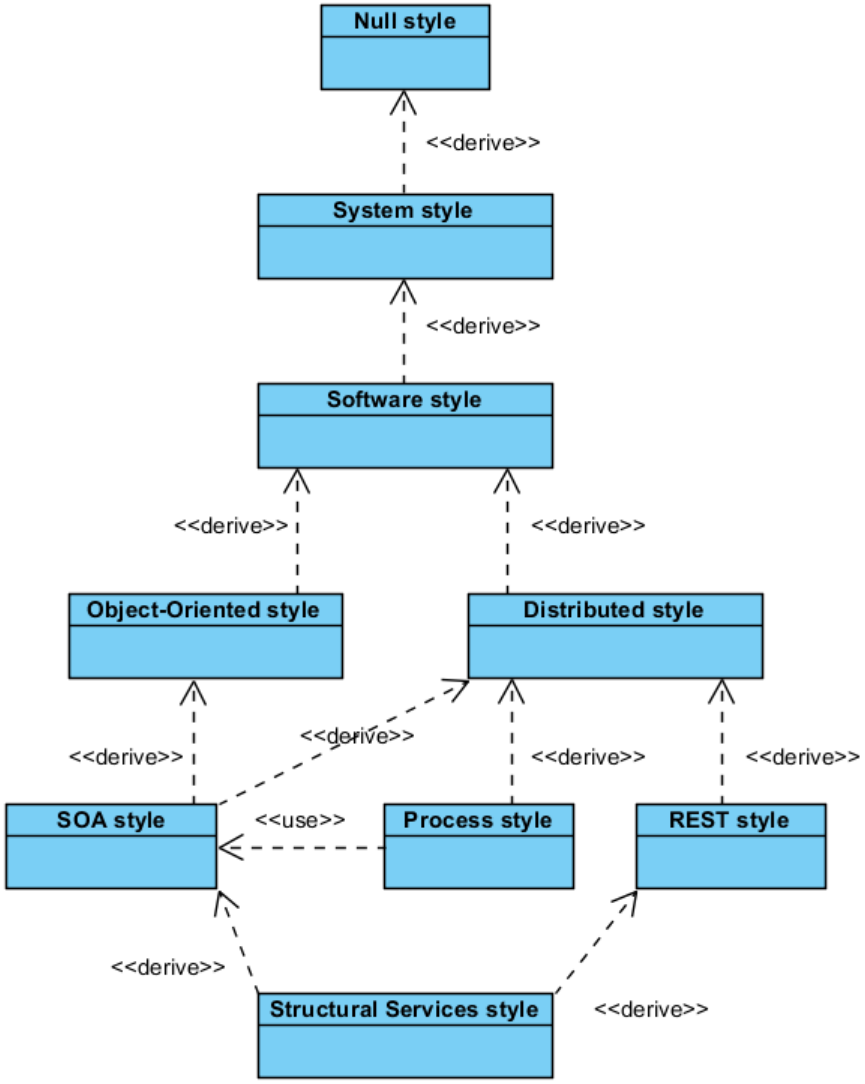


Figure 2. *A hierarchy of architectural styles*

Multiple derivations in Figure 2 are conceptual and apply to constraints. If a constraint applies to both architectural styles, the weakest (least restraining) is passed on to the derived style. The goal is to obtain the best of both styles by emphasizing flexibility. For example, regarding the Structural Service style:

- The SOA style allows any number of operations for each resource, whereas REST specifies that all resources must have the same operations. Therefore, we want SOA's flexibility in services.

- The REST style allows structured resources, whereas SOA does not. In this case, we want REST's flexibility in resources.

Therefore, we propose the Structural Services architectural style, in which a resource has both structured state (resources as components) and service behavior (set of operations), all defined by the resource designer. This means that a resource becomes similar to an object in object-oriented programming, but now in a distributed environment, with the same level of data interoperability that XML and JSON

provide. Table 1 summarizes the main characteristics of these architectural styles. For completeness, the characteristics of the Structural Services architectural style, described below, are also included.

Style	Brief description	Examples of characteristics or constraints
Null	No style	No constraints, no model elements identifiable
System	Identifies resources, services and processes (model of Figure 3)	Resources are discrete (individualized) Resources identified by references Services described in terms of reactions Service interact by stimuli (can be analog and continuous)
Software	All resources are software related	Resources are digital Stimuli are finite-sized and time-limited messages Resources obey a software development lifecycle
Object-Oriented	Resources and services follow the Object-Oriented paradigm	Resources are classes Services are interfaces References are pointers Reactions discretized into operations Polymorphism based on inheritance
Distributed	Interacting services have independent lifecycles	References cannot be pointers Type checking cannot be name-based Services and resources may use heterogeneous technologies
SOA	Style similar to Object-Oriented, but with Distributed constraints	Resources have no structure The interface of services is fixed No polymorphism Integration based on common schemas and ontologies
Process	Behavior modeled as orchestrations and choreographies	Services are the units of behavior A process is an orchestration of services Processes must obey the rules of choreographies
REST	Resources have structure but a fixed service	Client and server roles distinction Resources have a common set of operations Stateless interactions
Structural Services	Structured resources, variable services, minimal coupling, context awareness	Resources are structured (like REST) Services have variable interface (like SOA) Integration based on structural compliance and conformance (new feature) Functional and context-oriented interactions (new feature)

Table 1. *Main characteristics of architectural styles*

A Model of Resources and their Services

Figure 3 depicts our conceptual model of the basic entities involved in system interaction, namely, resources (systems), services (the functionality they offer) and processes (flows of service invocations).

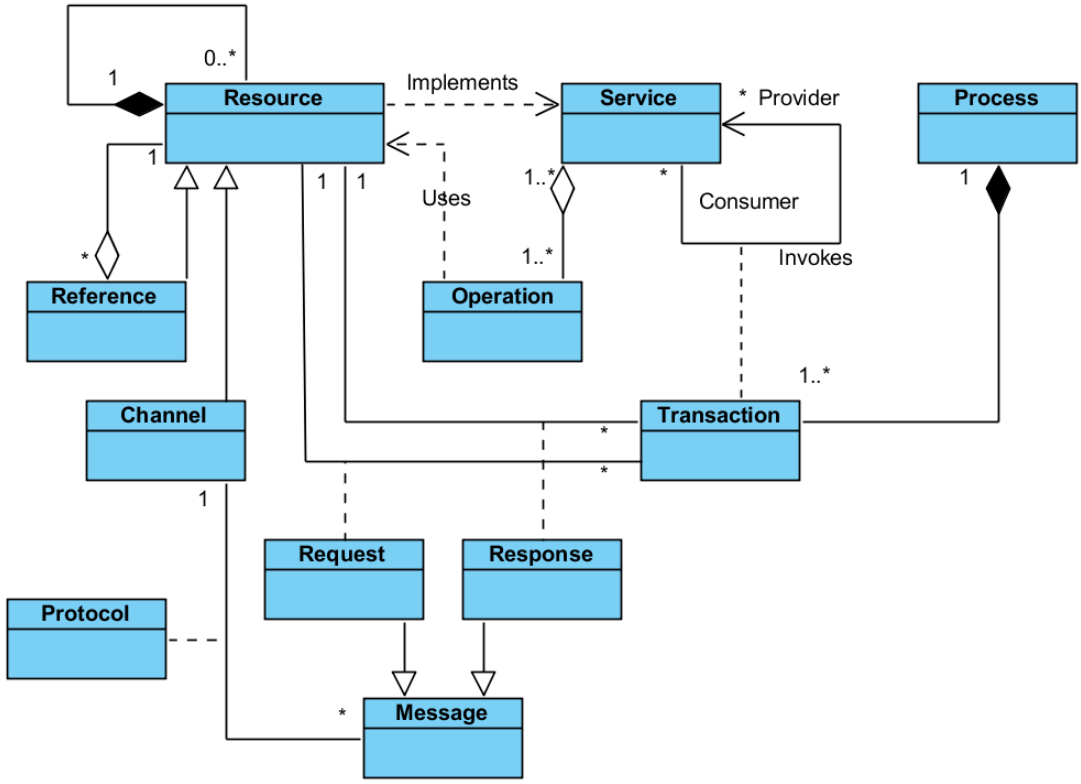


Figure 3. *A model of basic entities involved in system interaction*

This model can be described as follows:

- A *resource* is an entity of any nature (material, virtual, conceptual, noun, action, and so on) that embodies a meaningful, complete and discrete concept, makes sense by itself and can be distinguished from, although able to interact with, other entities. Resources are the main modeling entities and should express, as closely as possible, the entities in the problem domain. Physical resources range from simple sensors up to people, although the latter are represented in the computer-based realm by user interface devices. Non-physical resources typically correspond to software-based modules, endowed with interaction capabilities.

- Resources are discrete and distinct entities, atomic (indivisible whole) or structured (recursively composed of other resources, its components, with respect to which it performs the role of container). Each component can only have one direct container (yielding a strong composition model, in a tree shaped resource structure). Resources can also exhibit a weak composition model, by referring to each other by *references*. References are themselves resources and support the existence of a directed graph of resources, superimposed on the resource tree.

- Atomic resources can have state, either immutable or changeable. The state of a structured resource is recursively the set of the states of its components. Resources can migrate (move) from one container to another, changing the system's structure.

- Resources are self-contained and interact exclusively by sending *messages*, i.e., resources moved from the sender to the receiver. Resources are typically distributed and messages are exchanged through a *channel*, a resource specialized in relaying messages between interacting resources.

- Each resource implements a *service*, the set of operations supported by that resource and that together define its behavior (the set of reactions to messages that the resource exhibits).

- One resource plays the role of *consumer* and sends a *request* message (over the channel) to another resource, which plays the role of *provider* and executes the request, eventually sending a *response* message back to the consumer. This basic interaction pattern constitutes a *transaction*. More complex interactions can be achieved by composing transactions, either temporally (the consumer invokes the provider several times) or spatially (the consumer *A* invokes the provider *B*, which in turn invokes another provider *C*, and so on).

- A *process* is a graph of all the transactions that are allowed to occur, starting with a transaction initiated at some resource and ending at a transaction that neither provides a response nor initiates new transactions. A process corresponds to a use case of a resource and usually involves other resources, as transactions flow (including loops and recursion, eventually).

In Figure 3, association classes describe relationships (such as Transaction and the message Protocol) or roles, such as Message, which describes the roles (specialized by Request and Response) performed by resources when they are sent as messages. The interaction between resources has been represented as interaction between services, since the service represents the active part (behavior) of a resource and exposes its operations. When we mention resource interaction, we are actually referring to the interaction between the services that they implement.

In summary, the three topmost entities of this model can be characterized as:

- Resources entail structure, state and behavior.
- Services refer only to behavior, without implying a specific implementation.
- Processes are a view of the behavior sequencing and flow along services, which are implemented by resources.

Partial Interoperability with Compliance and Conformance

Usually, resources are made interoperable by design, i.e., conceived and implemented to work together, allowing the service of one to invoke the service of another. This can be seen in program development and distributed integration, using for example Web Services. In this case:

- Schemas are shared between interacting services, establishing coupling for all the possible values satisfying each schema, even if they are not actually used.

- REST also requires that data types (usually called media types) be standardized or previously agreed, when they are application specific.

- Searching for an interoperable service is done by schema matching with *similarity* algorithms (Jeong, Lee, Cho, & Lee, 2008) and ontology matching and mapping (Euzenat & Shvaiko, 2007). This does not ensure interoperability and manual adaptations are usually unavoidable.

The interoperability notion, as defined in this chapter, introduces a different perspective, stronger than similarity but weaker than commonality (resulting from using the same schemas and ontologies). The trick is to allow partial (instead of full) interoperability, by considering only the intersection between what the consumer needs and what the provider offers. If the latter subsumes the former, the degree of

interoperability required by the consumer is feasible, regardless of whether the provider supports additional capabilities or not.

Interoperability (of a consumer with a provider) entails the following properties:

- *Compliance* (Kokash & Arbab, 2009). The consumer must satisfy (*comply with*) the requirements established by the provider to accept requests sent to it, without which these cannot be honored.

- *Conformance* (Kim & Shen, 2007; Adriansyah, van Dongen, & van der Aalst, 2010). The provider must fulfill the expectations of the consumer regarding the effects of a request (including eventual responses), therefore being able to take the form of (*to conform to*) whatever the consumer expects it to be.

In full interoperability, the consumer can use all of the provider's capabilities. In partial compatibility, the consumer uses only a subset of those capabilities, which means that compliance and conformance need only be ensured for that subset.

These properties are not commutative (e.g., if *A* complies with *B*, *B* does not necessarily comply with *A*) but are transitive (e.g., if *A* complies with *B* and *B* complies with *C*, then *A* complies with *C*).
Figure 4 illustrates this model. A resource *A*, in the role of consumer, has been designed for full interoperability with resource *B*, in the role of provider. *A* uses only the capabilities that *B* offers and *B* offers only the capabilities that *A* uses. Let us assume that we want to change the provider of *A* to resource *X*, which has been designed for full interoperability with resource *Y*, in the role of consumer. The problem is that *A* was designed to interact with provider *B* and *X* was designed to expect consumer *Y*. This means that, if we use resource *X* as a provider of *A*, *B* is how *A* views provider *X* and *Y* is how *X* views consumer *A*.

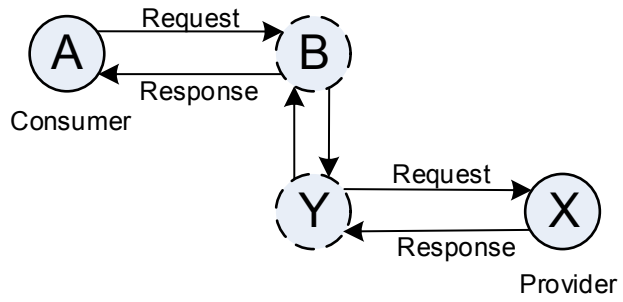


Figure 4. *Partial interoperability, based on compliance and conformance*

There are two necessary conditions to ensure that *A* is interoperable with *X*:

- *Compliance*: *B* must *comply with Y*. Since *A* complies with *B* and *Y* complies with *X*, this means that *A* complies with *X*, and therefore, *A* can use *X* as if it were *B*, as it was designed for.

- *Conformance*: *Y* must *conform to B*. Since *X* conforms to *Y* and *B* conforms to *A*, this means that *X* conforms to *A* and, therefore, *X* can replace (take the form of) *B* without *A* noticing it.

Partial interoperability has been achieved by subsumption, with the set of capabilities that *A* uses as a subset of the set of capabilities offered by *X*. This inclusion relationship, without changing characteristics, is similar in nature to polymorphism, used in many programming languages, but here it applied to a

distributed context. It constitutes the basis for transitivity in compliance and conformance, as well as the mechanism to reduce coupling between two resources to the minimum required by the application.

These properties are not commutative, since the roles of consumer and provider are different and asymmetric by nature. However, nothing prevents two interacting resources from switching roles in a symmetric way, by using and offering capabilities in a reciprocal fashion, which is typical of certain interaction protocols.

The SOA-REST Dichotomy

The SOA style is an extrapolation of the object-oriented style to distributed contexts, but the main goal of a low semantic gap is retained. This means that each SOA entity should model each entity in the problem domain as closely as possible, in a one-to-one mapping, and a small change in the problem should yield a small change in the SOA model. Each entity has its own interface and a consumer, to use the functionality of a provider, needs to know its different operations and semantics.

REST proponents (Pautasso, Zimmermann, & Leymann, 2008) contend that this an unacceptable coupling, hampering scalability, and that a consumer should only know the provider's link (URI, in Web terms), obtain a representation of it (an universal operation) and from then onwards follow the links contained in that representation, using only a fixed set of universal operations (supported by all the resources).

Fielding (2000) designated this as "hypermedia as the engine of application state" (HATEOAS). The basic idea is that the client (consumer) needs to know very little about the server (provider), since it only follows the links that the server provides, and that the server needs to know very little about the client, since it is the client that decides what link to follow. The goal is to minimize coupling and to maximize scalability.

It seems that REST achieves a lower coupling than SOA. However, this is an elusive statement, and both styles have advantages and disadvantages. To clarify this, consider the example described by Figure 5.

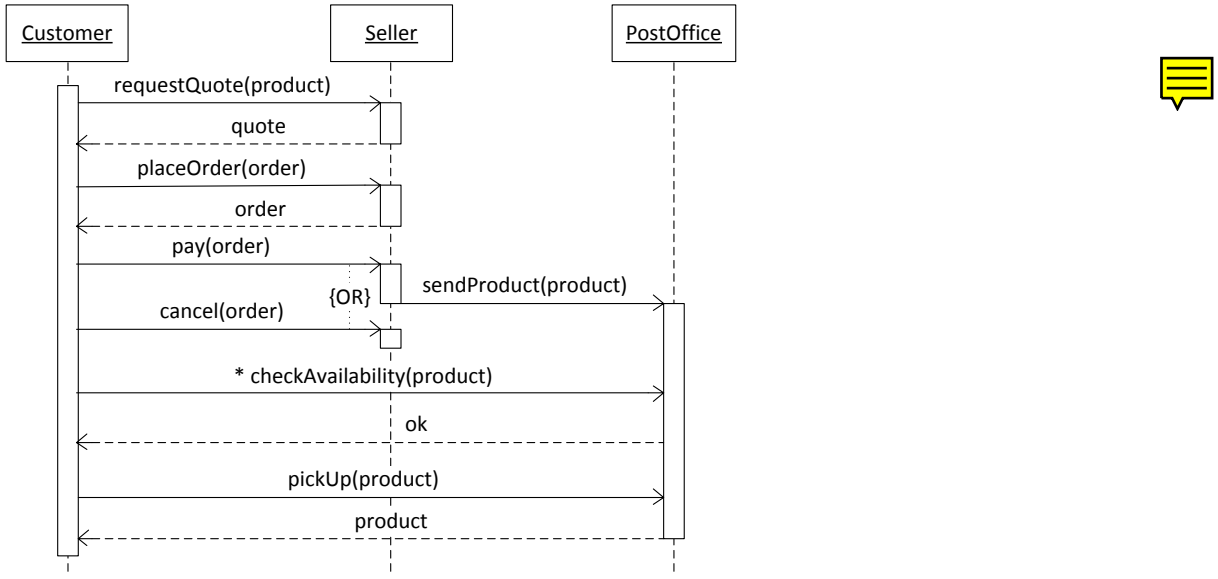
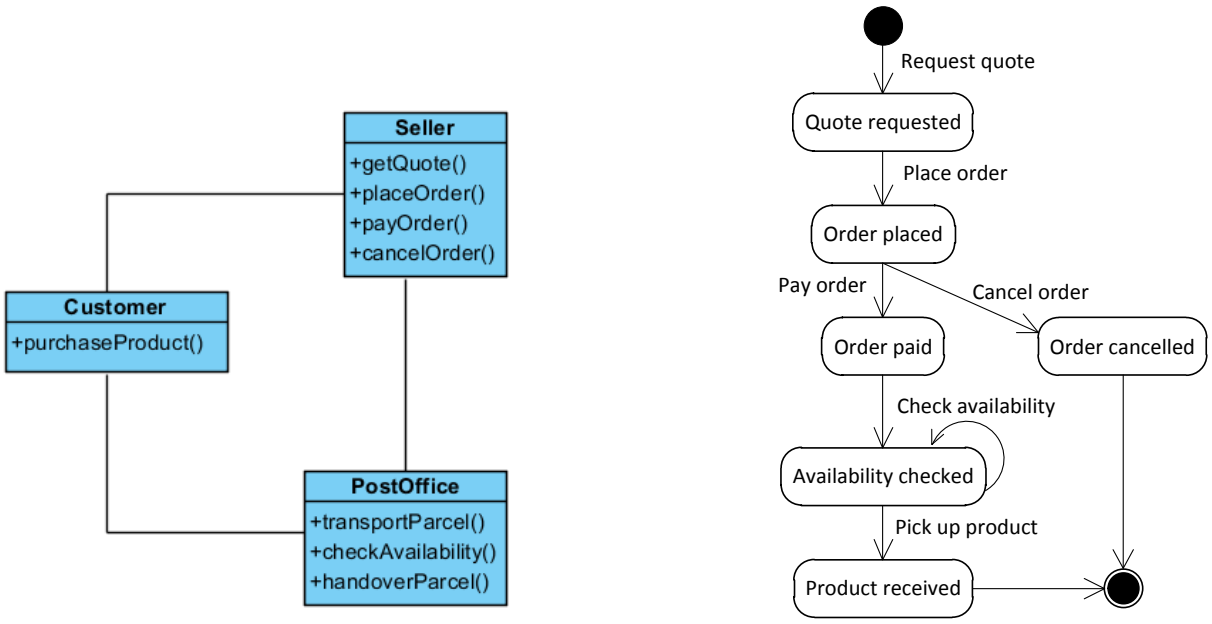


Figure 5. *An example of distributed resource interactions*

In this example, a customer wants to buy a product from a seller company, which delivers it by post. To obtain a cheaper transportation cost, the customer chooses to pick up the parcel with the product at the local post office instead of having it delivered at home. The customer first requests a quotation, then places an order and pays it, with the option of canceling the order. After that, the customer polls the mailbox periodically to check for a notification of arrival and, when that happens, picks up the parcel from the post office. This simple example could be modeled in SOA and REST styles as depicted by Figure 6a and 6b, respectively.



(a) (b)

Figure 6. *Modeling the example of Figure 5 in SOA (a) and REST (b) styles*

It is noticeable that, apart from details, the flow of activities of the client in SOA is dual of the flow in REST. Figure 6a (SOA) uses activity based flow, with activities in the nodes and state in between, whereas Figure 6b (REST) makes transitions from state to state, implemented by the activities in between. SOA is guided by behavior and REST by (representation of) state. In UML terminology, SOA uses a static class diagram (Figure 6a) as a first approach, to specify which resource types are used and to establish the set of operations provided by each, whereas REST starts with a state diagram (Figure 6b), without relevant concern about distinguishing which state belongs to which resource. In the end, they perform the same activities and go through the same states. This should not be a surprise since, after all, the original problem, described by Figure 5, is the same.

REST may seem more flexible, in the sense that, if the server changes the links it sends in the responses, the client will follow this change automatically by using the new links. The problem, however, is that this is not as general as it may seem, since the client must be able to understand the structure of the responses. It is not merely a question of following all the links in a response. To achieve this, REST imposes the constraint that returned representations use standardized or pre-agreed media types. Moreover, just stating that the format is XML or JSON is not enough. The actual set of names used (the schema, in fact) must be known by both client and server. This is why REST does not entail less coupling than SOA.

What REST does is to trade interface variability for structure variability, something that SOA lacks. REST cleanly separates the mechanism of traversing a graph from the processing of individual node

graphs (Meyer, 2000). Therefore, varying the structure allows changing the overall behavior without affecting the traversal mechanism. However, this requires that all nodes are treated alike, which means that all nodes must have the same interface, which requires that operations are first class resources, which in turn leads to a state diagram programming style, such as the one in Figure 6b.

The main problem with this is that the model is no longer guided by the static entities of the problem, in an object-oriented fashion, but rather by state, as an automaton. Most people will find it harder to model state transitions than static entities (classes). This is not a problem for simpler applications that can be organized in a CRUD (Create, Read, Update, Delete) approach, a natural way when structured state is the guiding concept, but more complex applications, in which behavior and information hiding (including state) are the primordial factors, are a different issue.

It turns out that many applications are simple and the technologies typically used to implement REST are simpler, lighter and (in many cases) cheaper than those used to implement SOA (namely, SOAP Web Services), which justifies the growing popularity of RESTful applications and their APIs. The level of resource coupling in REST, however, is not lower than that of service coupling in SOA, since both require that the schemas (media types) used are previously known by both interacting parties.

It should also be noted that SOA lacks support for structured resources. Services are seen as one level only, offering operations but hiding any internal structured state. Structure is a natural occurrence in most problem domains and in this respect REST is a better match. Therefore, our goal is to combine the service flexibility of SOA (which provides a low semantic gap in terms of behavior modeling) with the structural flexibility of REST (which leads to a low semantic gap in terms of modeling structural composition and state). We propose to achieve this goal by unifying the service and resource concepts in the Structural Services architectural style.

STRUCTURAL SERVICES: UNIFYING SERVICES AND RESOURCES

One of the main limitations of current integration technologies, based on either SOA or REST, is that they use the document concept as the foundation, with a data description language as the representation format and schema sharing as the interoperability mechanism. Other factors, such as a connectionless protocol (HTTP) and the lack of native support for binary data and contextual information, are limiting for many applications, although not impeditive.

Main Characteristics

To solve these limitations, we propose an architectural style (Structural Services), introduced in Figure 2 and Table 1, with the following main characteristics:

- The main modeling concept is the resource, which can be structured and is able to expose that structure. A resource can be identified by a globally accessible path, starting at a directory root or at some other previously known resource.

- Operations are first class resources but do not expose structure, only behavior. Data resources can be structured but expose state only (apart from basic getters and setters). Other resources are typically composed of operations and data resources.

- Resources are active and can respond to a message sent to it, by automatically choosing an operation adequate to process that message. If the receiver itself is an operation, that is the one invoked.

- A service is the exposed interface of a resource. This is the equivalent to a WSDL document, but also including component resources, not just operations.

- Resources are described by a distributed programming language, designed to be compiled and executed by a platform (server handler and an interpreter). Since resources are active (not passive data documents) and exhibit a service, a data description language is not enough.

- Messages are themselves resources and can include operations.

- Interoperability is based on structural compliance and conformance, not on schema sharing or predefined media types. This means that two interacting resources need to be interoperable only in the operations and/or data actually accessed (partial interoperability) and not on the entire definition of their services. This is a way of reducing coupling.

Although not essential to this architectural style, we have also included other characteristics (not detailed here, due to space limitations), such as:

- A dual representation format: text, a distributed programming language, and binary, obtained by compiling the text source and using TLV (Tag, Length and Value) binary markup (Dubuisson, 2000). Messages can include only source, binary with names and only binary. The latter is the most efficient, but is not self-descriptive. Therefore, the server has the capability of caching mappings between received messages and the parameters of the invoked operations, which the client can use (using a token returned by the server to identify a mapping) if the message is repetitive. The protocol supports retransmissions of messages with self-description information if for some reason the token is not valid.

- A layered protocol, with a generic message level separated from the transport level. There is no requirement for transport reliability. If the application needs it, and the protocol does not implement it, the application needs to do it with timeouts and exception handling.

- A context passing mechanism, backward and forward, achieved by passing two arguments (one functional, another contextual) to and from operations. Exceptions have been unified and included in the backward context mechanism. Support for this is provided by the distributed programming language and the message level protocol.

- Semantic description through rules, which are also first class resources, become active as soon as they are created and are reevaluated whenever a resource on which their condition depends has its state changed. If the condition is true, the action part (a statement) is executed. Rules can be used to raise exceptions when certain undesirable conditions become true.

Service discovery and composition stem naturally from these characteristics and are briefly described in the next section.

How can the Structural Services style help in solving the problem described in Figures 5 and 6? From the point of view of modeling actors, the SOA approach (Figure 6a) is more natural and leads to a smaller semantic gap with reality. But if we consider that the Seller may return a list of products which the Customer must analyze and choose from, then the easiest thing to do is to deal with that list in hypermedia fashion, invoking a known set of operations valid for all these products.

One of the advantages of Structural Services is the ability to lean more towards SOA or REST, according to modeling convenience, and mix several approaches and patterns in the same problem. In particular, we

should note that, in the Object-Oriented style, inheritance-based polymorphism is no more than a fixed set of operations for a subset of objects (those inheriting from a common base class, which defines the set of operations). In Structural Services, we can also have a fixed set of operations for a subset of resources, but now with distributed polymorphism, based on compliance and conformance.

It is important to acknowledge that the REST style requires a fixed set of operations, but without stating which operations should be in that set. The HTTP implementations use some of the HTTP verbs, but this is just one possibility. The Structural Services style allows a further step in flexibility, by opening the possibility of using several fixed set of operations in the same problem. We can have not only both extremes (one set for each resource, SOA style, or one set for all resources, REST style) but also any intermediate combination, with different sets of operations for different sets of resources, according to the polymorphism based on compliance and conformance.

Implementing the Structural Services Style

The basis for current integration technologies lies on data description languages such as XML and JSON, which merely describe data and their structure. JSON is much simpler than XML and, thanks to that, its popularity has been constantly increasing (Severance, 2012).
SOA is usually implemented by Web Services, with WSDL (a set of conventions on XML usage) to describe services at the interface level and SOAP as the message protocol (again, based on XML). The verbosity and complexity of WSDL and SOAP have progressively turned away developers in favor of the much simpler REST, which is usually implemented by messages directly on HTTP, using its main verbs as a fixed set of operations.
Since XML and JSON can only describe data, behavior is usually implemented by some generic programming language, which will not provide universal interoperability. At a higher level, BPEL (Juric & Pant, 2008) is platform agnostic, but it entails an unwieldy XML-based syntax that forces programmers to use visual programming tools that generate BPEL and increase the complexity stack. For example, a simple variable assignment, which in most programming languages would be represented as x=y, requires the following in BPEL:

```
<assign>
   <copy>
      <from variable="y" />
      <to variable="x" />
   </copy>
</assign>
```

The evolution of the dynamic nature of the Web, as shown by JavaScript and HTML5 (Lubbers, Albers, & Salim, 2010), hints that data description is not enough anymore and distributed programming is a basic requirement.

Current Web-level interoperability technologies are greatly constrained by the initial decision of basing Web interoperability on data (not services) and text markup as the main description and representation mechanism. This has had profound consequences in the way technologies have evolved, with disadvantages such as textual parsing overheads, full interoperability only (based on schema sharing) and cumbersome syntax (e.g., WSDL, BPEL) because everything must be described as data.

We could try to implement the Structural Services style using current technologies. For example, we could extend WSDL documents (with a structural section to support structured resources) or emulate variable set of operations on top of HTTP verbs. But these technologies and the corresponding set of tools were not conceived for this and compliance and conformance checks would just be too slow. The simplicity of REST would be lost and the complexity of SOA would be increased.

Therefore, we followed a different approach, by placing basic interoperability at the resource and service description levels, instead of merely at the data level. We defined and implemented a distributed programming language (SIL – Service Implementation Language), with a classic syntax style and an object-oriented look and feel, with the goal of providing native support for both data components and operations, as well as structural interoperability, based on structural compliance and conformance (Delgado, 2013). Next section presents an example.

This is the only way in which we can test the new features of Structural Services without the limitations of current technologies. Obviously, the goal is to provide a proof of concept and not to compete with established technologies or tools.

Each resource in SIL is represented by a SPID (*SIL Public Interface Descriptor*), which corresponds to a Web Service's WSDL but much more compact and able to describe both structure and operations. It is automatically obtained from the resource description itself by including only the public parts (public component resources and operation headings). Unlike XML or even JSON, there is no separate schema document to describe a resource.

We have developed a compiler for SIL based on ANTLR (Parr, 2013), which is able to produce the SPID automatically and to convert source to instructions and data in a binary format, using TLV as described above. An interpreter then executes the binary code (silcodes, similar to Java's bytecodes). The current implementation, in pure Java, is optimized for flexibility rather than performance and is roughly 50 times slower than a Java Virtual Machine (JVM). Much of that time is spent just on method dispatch, the mechanism used to execute the various silcodes. A C/C++ based interpreter would be faster but less portable and harder to develop and to maintain. Given the need for flexibility and control of implementation, we did not use a JVM as a compilation target.

We used a Jetty application server to support distribution, but any other server will do. In fact, we only need a handler for the protocol, HTTP or any other. For message exchange, we require only a transport level protocol. We implemented our own message protocol on top of that. We currently use WebSockets (Lubbers, Albers, & Salim, 2010), with a cache for automatic connection management, but any lower level message transport protocol will be enough, provided that it implements the level of reliability required by the application or the application provides that itself.

The Jetty server connects to a SIL server (to handle the SIL message level protocol) that hosts a resource directory for service discovery. This is a regular service, just like any other, which contains references to the SPIDs of the resources registered in it. This directory can be searched for a suitable service by supplying keywords and/or a SPID as required by the client. The directory then searches for these keywords in the registered SPIDs and performs a structural interoperability check to ensure that the returned references to SPIDs are conformant to the SPID used in the search. Similarity ranking (Formica, 2008), as an aid for semi-automatic service search, is not yet supported.

Service composition is done simply by defining a new resource that wraps a set of references to the resources with the existing services and implements the composed behavior with new operations that invoke the existing services through those references. In essence, this is nothing more that classical object composition, but now with distributed interoperability. Given the support for both behavior and structure, the composition style can vary, from a more operation-centric to a more data-centric approach.

An Example

This section illustrates the distributed programming language that we have developed to implement the Structural Services style.

Program 1 shows a simple example of resources described in SIL, involving sensors in the context of the Internet of Things (Gubbi, Buyya, Marusic, & Palaniswami, 2013). It includes a temperature controller (`tempController`), with a list of references to temperature sensors with history (`tempSensorStats`). Each of these has a reference to a remote temperature sensor (`tempSensor`). The lines at the bottom illustrate how these resources can be used and should be included in some resource that uses them.

```
tempSensor: spid {  // descriptor of a temperature sensor
  getTemp: operation (-> [-50.0 .. +60.0]);
};

tempSensorStats: { // temperature sensor with statistics
  sensor: @tempSensor;      // reference to sensor (can be remote)
  temp: list float;         // temperature history
  startStats<||;            // spawn temperature measurements
  getTemp: operation (-> float) {
    reply sensor@getTemp<--; // forward request to sensor
  };
  getAverageTemp: operation ([1 .. 24] -> float) {
    for (j: [temp.size .. temp.size-(in-1)])
      out += temp[j];
    reply out/in; // in = number of hours
  };
  private startStats: operation () {  // private operation
    while (true) {
      temp.add<-- (getTemp<--); // register temperature
      wait 3600;     // wait 1 hour and measure again
    }
  }
};

tempController: { // controller of several temperature sensors
  sensors: list @tempSensorStats;//list of references to sensors
  addSensor: operation (@tempSensor) {
    t: tempSensorStats;    // creates a tempSensorStats resource
    t.sensor = in;              // register link to tempSensor
    sensors.add<-- @t;     // add sensor to list
  };
  getStats: operation (-> {min: float; max: float; average: float})
  {
    out.min = sensors[0]@getTemp<--;
    out.max = out.min;
    total: float := out.min;     // initial value
    for (i: [1 .. sensors.length-1) {  // sensor 0 is done
      t: sensors[i]@getTemp<--; // dereference sensor i
      if (t <out.min) out.min = t;
      if (t >out.max) out.max = t;
      total += t;
    };
    out.average = total/sensors.length;
    reply; // nothing specified, returns out
  }
```

```
};

// How to use the resources
// tc contains a reference to tempController
tc@addSensor<-- ts1; // reference to a tempSensor resource
tc@addSensor<-- ts2; // reference to a tempSensor resource
x: tc@sensors[0]@getAverageTemp<-- 10;  // average last 10 hours
```

Program 1. *Describing and using resources, using SIL (Service Implementation Language)*

This program can be described in the following way:

- The temperature sensor (`tempSensor`) is remote and all we have is its SPID, the equivalent to a Web Service's WSDL. For example, the SPID of `tempSensorStats` can be expressed by the following lines:

```
tempSensorStats: spid { // temperature sensor with statistics
   sensor: @tempSensor;     // reference to sensor (can be remote)
   temp: list float;        // temperature history
   getTemp: operation (-> float);
   getAverageTemp: operation ([1 .. 24] -> float);
}
```

- Resources and their components are declared by a name, a colon and a resource type, which can be primitive, such as `integer`, a range (e.g., [1 .. 24]), `float` or user defined (enclosed in braces, i.e., "{. . .}"). There are some resemblances to JSON, but component names are not strings and operations are supported as first class resources.

- The definition of a resource is similar to a constructor. It is executed only once, when the resource is created, and can include statements. This is illustrated by the statement "`startStats<||`" in `tempSensorStats`. Actually, this is an asynchronous invocation ("`<||`") of private operation `startStats`, which is an infinite loop registering temperature measurements every hour. Asynchronous invocations return a *future* (Schippers, 2009), which will be later replaced by the returned value (in this example, the returned future is ignored). Synchronous invocation of operations is done with "`<--`", followed by the argument, if any.

- Operations have at most one argument, which can be structured (with "{. . .}"). The same happens with the operation's reply value, as illustrated by operation `getStats`. Inside operations, the default names of the argument and the value to return are `in` and `out`, respectively. The heading of operations specifies the type of the argument and of the reply value (inside parentheses, separated by "`->`").

- References to resources (indicated by the symbol "`@`") are not necessarily URIs, not even strings. They can also be structured and include several addresses, so that a resource in a network (e.g., the Internet) can reference another in a different network, with a different protocol (e.g., a sensor network). It is up to the nodes and gateways of the network, supporting these protocols, to interpret these addresses so that transparent routing can be achieved, if needed.

- Resource paths, to access resource components, use dot notation, except if the path traverses a reference, in which case a "`@`" is used. For example, the path used in the last line of Program 1 computes the average temperature, along the last 10 hours, in sensor 0 of the controller.

Program 2 illustrates the compliance and conformance concepts, by providing two additional temperature sensors (`weatherSensor` and `airSensor`) in relation to Program 1. Only the additional and relevant parts are included here, with the rest as in Program 1.

```
tempSensor: spid {
   getTemp: operation (->[-50.0 .. +60.0]);
};

weatherSensor: spid {
   getTemp: operation (->[-40.0 .. +50.0]);
   getPrecipitation: operation (-> integer);
};

airSensor: spid {
   getTemp: operation (->[-40.0 .. +45.0]);
   getHumidity: operation (-> [10 .. 90]);
};

// tc contains a reference to tempController
tc@addSensor<-- ts1; // reference to a tempSensor resource
tc@addSensor<-- ts2; // reference to a tempSensor resource
tc@addSensor<-- as1; // reference to an airSensor resource
tc@addSensor<-- ws1; // reference to an weatherSensor resource
tc@addSensor<-- ws2; // reference to an weatherSensor resource
tc@addSensor<-- as2; // reference to an airSensor resource

x: tc@sensors[0]@getAverageTemp<-- 10;      // average last 10 hours
s: {max: float; average: float};
s = tc@getStats<--;     // only max and average are assigned to s

temp: [-50.0 .. +50.0];
temp = ws1@getTemp <--; // complies. Variability ok
temp = ts1@getTemp <--; // does not comply. Variability mismatch
```

Program 2. *Example of partial interoperability, with structural compliance and conformance*

In Program 2:

- `weatherSensor` and `airSensor` conform to `tempSensor`, since they offer the same operation and the result is within the expected variability. This means that they can be used wherever a `tempSensor` is expected, which is illustrated by adding all these types of sensors to `tempController` (through `tc`, a reference to it) as if they were of type `tempSensor`. Non relevant operations are ignored.

- The result of invoking the operation `getStats` on `tempController` is a resource with three components (as indicated in Program 1), whereas "`s`" has only two. However, the assignment is still possible. Structural interoperability ignores the extra component.

- The last statement triggers a compliance check by the compiler, which issues an error. The variability of the value returned by operation `getTemp` in `tempSensor` (referenced by `ts1`) is outside the variability range declared for component `temp`.

Discussion and Rationale

The expressive power of XML and of XML Schema is greater than that of JSON and JSON Schema, although at the expense of simplicity. The SOA architectural style has a higher abstraction level (lower semantic gap in modeling reality) and provides better design support than REST. However, the XML and SOA worlds have greater complexity, are harder to master and raise the question of whether all the features provided by a more powerful but more complex specification are really needed or even actually used.

The growing popularity of JSON and REST seem to indicate that developers will gladly trade expressive power and design support for simplicity. After all, the limiting factor in complex system design is the inability of humans to deal with too many issues at the same time. Complexity is one of the developer's worst enemies. JSON and REST may not be as expressive or generic as XML and SOA, but they also work and are simpler and more pragmatic, although applications need be dressed in a CRUD (data-oriented) perspective to better fit the REST architectural style (an easier task in simpler applications).

This dichotomy (expressive power versus simplicity) bears strong resemblances with the debate that occurred 30 years ago, in the computer architecture realm. In the beginning of the 80's, complexity was on the rise. The instruction set architectures of server processors were being endowed with more and more features, with the goal of making them more and more powerful so that a better support to the software could be provided.

Unfortunately, instructions more complex meant hardware more complex and longer circuit delays. Even simpler instructions began suffering with this and taking proportionately more time to execute. Researchers started to note not only this but, above all, that the complex instructions were actually seldom used, if at all (Patterson & Ditzel, 1980). Due to complexity and semantic gaps, compilers were not able to benefit from complex instructions and were generating, in most cases, only simpler instructions. It should be stressed that computer architects were not programming and programmers were not designing computers.

A new generation of computers appeared, dubbed RISC (for Reduced Set Instruction Computers), with only a handful of simple instructions, architecture completely open to the compiler and above all designed together with the compiler. These machines were shown to perform better than conventional computers, or CISC (Complex Instruction Set Computers), due to the emphasis on simplicity and to a good match between software and hardware.

By the early nineties, it became fashionable to have an instruction set as small as possible and RISC manufacturers flourished. However, RISCs didn't win the war and CISCs didn't disappear. CISCs learned their lesson from RISCs and, without dogmas, asserted that there is nothing wrong in having complexity in the hardware as long as it is truly useful and does not slow down simple instructions. The secret lies in balancing and matching the capabilities of technology with the requirements of applications.

Establishing a parallel with integration technologies, JSON and REST seem to perform the role of RISC, whereas XML and SOA seem to perform the role of CISC. Once again, it seems that the best solution is to combine the best features of both alternatives, in an attempt to emphasize the advantages and to minimize the limitations. This is the rationale behind the conception of the Structural Services architectural style, trying to reap the benefits from both SOA and REST.

We contend that the Structural Services architectural style provides a better solution to the integration problem than SOA or REST, essentially due to the following reasons:

- Providing native support for both resource-specific behavior and structure entails the possibility of adopting a more behavior-centric or data-centric approach, tuning it according to the

characteristics of the problem to solve. The solution can be mixed and several combinations of the pure SOA and REST styles can be used in the same application.

- Building interoperability on compliance and conformance avoids the problem of having to define schemas as separate documents and to agree upon them beforehand. As long as compliance and conformance hold, any two resources can interoperate, even if they were designed unbeknownst to each other.

- Adopting the resource, with the service it implements, as the basic entity description unit, avoids having to describe active entities (exhibiting behavior) on top of passive data description languages, such as XML and JSON, which are too low level for this purpose and yield complex descriptions with cumbersome syntax, such as BPEL.

This is why we decided to design SIL not as a passive data description language but as an active service implementation language, with the corresponding execution platform (Delgado, 2013). This has a profound impact in expressiveness and simplicity, because now resources can be described directly as the active entities they implement (services) instead of indirectly as low level data descriptions that lead to the verbosity and complexity typically found in XML-based documents, such as WSDL and BPEL.

This unified vision of resources and services can also be illustrated by considering how service discovery, or the equivalent of UDDI, is done in SIL. There is a directory service, with which resources available to provide services can be registered. This is a regular service, not a special mechanism, and can register resources within a single computing node or remotely available. The directory maintains a list of references to registered resources. When we want to search for a service, we supply the directory with the SPID that describes the service that we want and the directory returns a list of references to all the registered resources that conform to that SPID. Conformance includes context information, so that we can specify both functionality and non-functional aspects and constraints, such as security mechanisms, SLA, cost, and so on. This is how the references used in Program 1 can be obtained.

The most used solutions to implement service behavior are external programming languages (e.g., a Web Service or a REST resource implemented in Java) and WS-BPEL, an XML-based language. Services are described separately (e.g., in a WSDL file). Several languages and specifications are needed to implement the full solution. SIL includes its execution platform and is complete in the sense that one single language is enough to describe, implement and execute services, with their state structure and behavior. A complete distributed application can be implemented in SIL. Nevertheless, interoperability with other languages (currently, only Java) is supported, so that primitive operations can be implemented not in SIL but in a native language. This uses a mechanism similar to what current technologies use, with a mapping between the data types constructs in SIL and the corresponding ones in other languages. SIL has the advantage of being object-oriented and therefore constitutes a good match for object-oriented languages.

To avoid inheriting the constraints of current technologies, SIL is a new solution and not an evolutionary approach. The lack of compatibility can be overcome by establishing a migration path along the following lines:

- Co-existence with current technologies. The SIL server receives binary messages, which can be SIL resources or documents in XML or JSON. The SIL handler first checks if this is an identifiable SIL message (in TLV format). If not, the full message can then be inspected by other handlers, which implement current technologies. The SIL server does not depend on any particular transport protocol, relying solely on message delivery. Any existing server can be used, based on HTTP, WebSockets or any other protocol. In fact, several servers can be used

simultaneously, receiving messages that are handed over to the message handlers that are able to process them.

- Conversion from and to existing formats. SIL has been conceived to support the features provided by XML, JSON and the corresponding schema languages, which means that any XML, WSDL or JSON file can be converted to SIL, allowing current clients to use a service in a SIL server if a suitable converter in inserted in the path. The inverse can also be done, although it may require some extra work to express the features of SIL that do not exist in XML or WSDL, such as context information.

Getting back to the scenarios of Figure 1, we note that different integration solutions are used today in different scenarios, because these have evolved from different problems and naturally yielded different solutions, from basic HTTP and HTML to Web Services, RESTful APIs and specific protocols, APIs and models.

In retrospect, we can see that they are all instances of the same problem: how to integrate two resources by exchanging messages. Using service/resource description languages, such as SIL, instead of data description languages, such as XML and JSON, would make all these integration problems much less heterogeneous and easier to develop and to maintain.

For example, a web server would share a common base with a cloud, a grid or a sensor network. If everything is a resource exhibiting a service, then web server and API requests would all be messages and the common ground for resource interaction would be several layers up in the interoperability ladder, in particular if semantics or even pragmatics (behavior) are introduced. This is the vision that motivated the conception of the Structural Services architectural style.

FUTURE RESEARCH DIRECTIONS

Currently, there is a partial implementation of the SIL language and environment with basic functionality (Delgado, 2013). Future work will be carried out along the following main lines.

Compliance and conformance are basic concepts in interoperability and can be applied to all domains and levels of abstraction and complexity. Although work exists on its formal treatment in specific areas, such as choreographies (Adriansyah, van Dongen, & van der Aalst, 2010), an encompassing and systematic study needs to be conducted on what is the formal meaning of compliance and conformance. The compliance and conformance algorithms are implemented in the compiler but these need to be optimized in the binary version, to increase the performance of dynamic type checking.

SIL supports semantics through rule resources (set of condition-then-action statements), which become active upon creation in the scope of their container resource and are re-evaluated each time a component affecting one of the conditions is changed. It is up to the compiler to establish dependencies between conditions and components. This mechanism needs to be optimized.

In spite of all its diversity, the Web is a homogenous network (all web devices use HTTP as the underlying protocol and text-based data description languages), with a very reliable backbone. This is not particularly suitable to newer networks and environments, such as the Internet of Things (Luigi, Iera, & Morabito, 2010) and mesh network applications (Benyamina, Hafid, & Gendreau, 2012), for which dynamic and adaptive protocols, as well as messages with binary formats, provide a better match. The integration problem acquires completely new proportions and new solutions need to be envisaged, with particular emphasis on reliability in the context of *adhoc* networks.

A quantitative comparison assessment of the SIL platform versus SOA Web Services and RESTful solutions needs to be carried out in several contexts, including Internet of Things applications and non-IP networks.

CONCLUSION

Current integration technologies for distributed systems are essentially supported, in one way or another, on textual data description languages (XML and JSON) and on a stateless and connectionless protocol (HTTP). These have been conceived initially for human-level interaction and are not as adequate to the current range of global integration problems, now dominated by computers and involving binary data and real-time communications, in heterogeneous environments and networks. These technologies today are heavily influenced by the decisions taken for a class of problems (hypermedia interoperability at the human level) that is not dominant anymore.

The result is that, by current application requirements, XML and JSON are too low level (forcing all kinds of specifications, such as WDSL, SOAP, BPEL and semantic descriptions, to verbose and inefficient mappings to their syntax). The foundation concept should be the active resource with a service, not a passive data document. HTTP is too high level, with many application-level decisions taken in the context for which it was designed, but not adequate to current contexts. The protocol should be layered and readily support binary data, a generic message-level protocol and asynchronous, full-duplex communication.

We proposed a new architectural style, Structural Services, with features conceived to minimize these problems, by unifying the concepts of service and resource. In particular, we have shown that the SOA and REST architectural styles are dual of each other, with complementary characteristics, which combined together provide an enhanced support for a broader range of applications. In Structural Services, resources expose structure and services can offer a varied set of operations.

The fundamental advantage of the Structural Services style is that it not only supports both SOA and REST extremes, with one set of operations for each resource or for all resources, but also any intermediate combination, with fixed set of operations for only the resources that conform to a given specification and that should be treated in the same way. This means that different sets of operations can be used, instead of just one as REST imposes, and that we can choose the combinations that provide a better modeling match to the problem entities and/or to the architectural solution envisaged.

An implementation of this style could be conceived with Web Services, for example, by including a structural section in WSDL documents, but this would add up to the already very complex Web Services technology. Instead, we opted to use a resource and service description language that natively supports both data components and operations, with structural interoperability, supported on structural compliance and conformance, which have been informally introduced.

The goal is not to replace SOA or REST, but rather to evaluate what can be gained by using a technology that natively supports the architectural style that we have proposed, instead of increasing the complexity resulting from the mismatch between this style and classical technologies such as HTTP, XML and JSON.

REFERENCES

Adamczyk, P., Smith, P., Johnson, R., & Hafiz, M. (2011). REST and Web services: In theory and in practice. In Wilde, E., & Pautasso, C. (Eds) *REST: from research to practice* (pp. 35-57). New York, NY: Springer.

Adriansyah, A., van Dongen, B., & van der Aalst, W. (2010). Towards robust conformance checking. In Muehlen, M. & Su, J. (Eds.) *Business Process Management Workshops* (pp. 122-133). Berlin Heidelberg: Springer.

Aggarwal, C., & Han, J. (2013). A Survey of RFID Data Processing. In Aggarwal, C. (Ed.) *Managing and Mining Sensor Data* (pp. 349-382). Springer US.

Armbrust, M., *et al*. (2010). A view of Cloud computing. *Communications of the ACM*, 53(4), 50-58.

Athanasopoulos, G., Tsalgatidou, A., & Pantazoglou, M. (2006). Interoperability among Heterogeneous Services, in *International Conference on Services Computing* (pp. 174-181).Piscataway, NJ: IEEE Society Press.

Benyamina, D., Hafid, A., & Gendreau, M. (2012). Wireless Mesh Networks Design – A Survey, *IEEE Communications Surveys & Tutorials*, 14(2), 299-310.

Berners-Lee, T. (1999). *Weaving the web: the original design and ultimate destiny of the World Wide Web by its inventor*. New York, NY:Harper Collins Publishers.

Berre, A. *et al*. (2007). The ATHENA Interoperability Framework. In Gonçalves, R., Müller, J., Mertins, K., & Zelm, M. (Eds.) *Enterprise Interoperability II* (pp. 569-580). London, UK: Springer

Bravetti, M., & Zavattaro, G. (2009). A theory of contracts for strong service compliance, *Journal of Mathematical Structures in Computer Science*, 19(3), 601-638.

Bruno, R., Conti, M., &Gregori, E. (2005). Mesh networks: commodity multihop ad hoc networks. *IEEE Communications Magazine*, 43(3), 123-131.

Castillo, P., et al. (2013). Using SOAP and REST web services as communication protocol for distributed evolutionary computation. *International Journal of Computers & Technology*, 10(6), 1659-1677.

Chen, D. (2006). Enterprise interoperability framework. In Missikoff, M., De Nicola, A. and D'Antonio F. (Eds.) *Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability*. Berlin, Heidelberg: Springer-Verlag.

Crane, D., & McCarthy, P. (2008).*Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Berkely, CA:Apress.

Delgado, J. (2013). Service Interoperability in the Internet of Things. In Bessis, N. et al (Eds.) *Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence* (pp. 51-87). Berlin Heidelberg: Springer.

Diaz, G., & Rodriguez, I. (2009). Automatically deriving choreography-conforming systems of services. In *IEEE International Conference on Services Computing* (pp. 9-16). Piscataway, NJ: IEEE Society Press.

Dillon, T., Wu, C., & Chang, E. (2007). Reference architectural styles for service-oriented computing. In Li, K. et al. (Eds.) *IFIP International Conference on Network and parallel computing* (pp. 543–555).Berlin, Heidelberg:Springer-Verlag.

Dubuisson, O. (2000).*ASN.1 Communication Between Heterogeneous Systems*. San Diego, CA: Academic Press.

Erl, T. (2005).*Service-oriented architecture: concepts, technology and design*. Upper Saddle River, NJ: Pearson Education.

Erl, T. (2008).*SOA: Principles of Service Design*. Upper Saddle River, NJ: Prentice Hall PTR.

Erl, T., Balasubramanians, R., Pautasso, C., & Carlyle, B. (2011). *Soa with rest: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. Upper Saddle River, NJ: Prentice Hall PTR.

EIF (2010). European Interoperability Framework (EIF) for European Public Services, Annex 2 to the Communication from the Commission to the European Parliament, the Council, the European Economic and Social Committee and the Committee of Regions 'Towards interoperability for European public services. Retrieved September 12, 2013, from http://ec.europa.eu/isa/documents/isa_annex_ii_eif_en.pdf

El Raheb, K. *et al*. (2011). Paving the Way for Interoperability in Digital Libraries: The DL.org Project. In Katsirikou, A., & Skiadas, C. (Eds.) *New Trends in Qualitative and Quantitative Methods in Libraries* (pp. 345-352). Singapore: World Scientific Publishing Company.

Euzenat, J., & Shvaiko, P. (2007). *Ontology matching*. Berlin: Springer.

Fernando, N., Loke, S., & Rahayu, W. (2013). Mobile Cloud computing: A survey. *Future Generation Computer Systems*, 29(1), 84-106.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California at Irvine, CA.

Fielding, R., & Taylor, R. (2002). Principled Design of the Modern Web Architecture, *ACM Transactions on Internet Technology*, 2(2), 115-150.

Formica, A. (2008). Similarity of XML-schema elements: A structural and information content approach. *The Computer Journal*, 51(2), 240-254.

Galiegue, F., & Zyp, K. (Eds.) (2013). *JSON Schema: core definitions and terminology*. Internet Engineering Task Force Retrieved Sept. 12, 2013, from https://tools.ietf.org/html/draft-zyp-json-schema-04

Gottschalk, P., & Solli-Sæther H. (2008). Stages of e-government interoperability. *Electronic Government, An International Journal*, 5(3), 310–320.

Graydon, P. *et al*. (2012). Arguing Conformance. *IEEE Software*, 29(3), 50-57.

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7), 1645–1660.

Hartenstein, H., & Laberteaux, K. P. (2008). A tutorial survey on vehicular ad hoc networks. *IEEE Communications Magazine*, 46(6), 164-171.

Haslhofer, B., & Klas, W. (2010). A survey of techniques for achieving metadata interoperability. *ACM Computing Surveys*, 42(2), 7:1-37.

Henkel, M., Zdravkovic, J., & Johannesson, P. (2004). Service-based Processes– Design for Business and Technology. In *International Conference on Service Oriented Computing* (pp. 21-29).New York, NY: ACM Press.

Holdener III, A. (2008). *Ajax: The Definitive Guide*.Sebastopol, CA: O'Reilly Media.

Hughes, D., Coulson, G., & Walkerdine, J. (2010). A Survey of Peer-to-Peer Architectures for Service Oriented Computing. In Exarchakos, G., Li, M., & Liotta, A. (Eds.) *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications* (pp. 1-19). Hershey, PA: IGI Global.

ISO (2011) *CEN EN/ISO 11354-1, Advanced Automation Technologies and their Applications, Part 1: Framework for Enterprise Interoperability*. Geneva, Switzerland: International Organization for Standardization.

ISO/IEC/IEEE (2010) Systems and software engineering – Vocabulary. *International Standard ISO/IEC/IEEE 24765:2010(E), First Edition*, p. 186. Geneva, Switzerland: International Organization for Standardization.

Jardim-Goncalves, R., Agostinho C., & Steiger-Garcao, A. (2012). A reference model for sustainable interoperability in networked enterprises: towards the foundation of EI science base. *International Journal of Computer Integrated Manufacturing*, 25(10), 855-873.

Jardim-Goncalves, R., *et al*. (2013). Systematisation of interoperability body of knowledge: the foundation for enterprise interoperability as a science. *Enterprise Information Systems*, 7(1), 7-32.

Jeong, B., Lee, D., Cho, H., & Lee, J. (2008) A novel method for measuring semantic similarity for XML schema matching, *Expert Systems with Applications*, 34, 1651–1658

Johnston, A., Yoakum, J., & Singh, K. (2013). Taking on WebRTC in an Enterprise. *IEEE Communications Magazine*, 51(4), 48-54.

Juric, M., & Pant, K. (2008).*Business Process Driven SOA using BPMN and BPEL: From Business Process Modeling to Orchestration and Service Oriented Architecture*. Birmingham, UK:Packt Publishing.

Keyes, J. (2013). *Bring Your Own Devices (BYOD) Survival Guide*. CRC Press.

Khadka, R. *et al*. (2011). Model-Driven Development of Service Compositions for Enterprise Interoperability. In van Sinderen, M., & Johnson, P. (Eds.), *Enterprise Interoperability* (pp. 177-190).Berlin, Heidelberg: Springer-Verlag.

Kim, D., & Shen, W. (2007). An Approach to Evaluating Structural Pattern Conformance of UML Models. In *ACM Symposium on Applied Computing* (pp. 1404-1408), New York, NY: ACM Press.

Kokash, N., & Arbab, F. (2009). Formal Behavioral Modeling and Compliance Analysis for Service-Oriented Systems. In Boer, F., Bonsangue, M., & Madelaine, E. (Eds.) *Formal Methods for Components and Objects* (pp. 21-41).Berlin, Heidelberg: Springer-Verlag.

Läufer, K., Baumgartner, G., & Russo, V. (2000). Safe Structural Conformance for Java. *Computer Journal*, 43(6), 469-481. Oxford, UK: Oxford University Press.

Li, W., & Svard, P. (2010). REST-based SOA Application in the Cloud: A Text Correction Service Case Study. In *6th World Congress on Services* (pp. 84-90). Piscataway, NJ: IEEE Society Press.

Liu, Y., Rong, Z., Jun, C., & Ping, C. Y. (2011). Survey of Grid and Grid Computing. In *International Conference on Internet Technology and Applications*, (pp. 1-4). Piscataway, NJ: IEEE Society Press.

Loreto, S., & Romano, S. (2012). Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. *IEEE Internet Computing*, 16(5), 68-73.

Loutas, N., Kamateri, E., Bosi, F., & Tarabanis, K. (2011). Cloud computing interoperability: the state of play. In Lambrinoudakis, C., Rizomiliotis, P. and Wlodarczyk T. (Eds.) *International Conference on Cloud Computing Technology and Science* (pp. 752-757). Piscataway, NJ: IEEE Society Press.

Loutas, N., Peristeras, V., &Tarabanis, K. (2011). Towards a reference service model for the Web of Services, *Data & Knowledge Engineering*, 70, 753–774.

Lubbers, P., Albers, B., & Salim, F. (2010).*Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. New York, NY: Apress.

Marjanovic, O. (2005). Towards IS supported coordination in emergent business processes. *Business Process Management Journal*, 11(5), 476-487.

Meyer, B. (2000).*Object-Oriented Software Construction*. Upper Saddle River, NJ: Prentice Hall.

Mooij, A., & Voorhoeve, M. (2013). Specification and Generation of Adapters for System Integration. In van de Laar, P., Tretmans, J. &Borth, M. (Eds.) *Situation Awareness with Systems of Systems* (pp. 173-187). New York, NY: Springer.

Muracevic, D., & Kurtagic, H. (2009). Geospatial SOA using RESTful web services. In *31st International Conference on Information Technology Interfaces* (pp. 199-204). Piscataway, NJ: IEEE Society Press.

Mykkänen, J., & Tuomainen, M. (2008). An evaluation and selection framework for interoperability standards.*Information and Software Technology*, 50, 176–197.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Raleigh, NC: Pragmatic Bookshelf.

Patterson, D., & Ditzel, D. (1980). The case for the reduced instruction set computer, *ACM SIGARCH Computer Architecture News*, 8(6), 25-33.

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. "big"' web services: making the right architectural decision, In *17th International conference on World Wide Web* (pp. 805-814).New York, NY: ACM Press.

Pautasso, C. (2009).RESTful Web service composition with BPEL for REST.*Data & Knowledge Engineering*, 68(9), 851-866.

Peng, Y., Ma, S., & Lee, J. (2009). REST2SOAP: A framework to integrate SOAP services and RESTful services. In *IEEE International Conference on Service-Oriented Computing and Applications* (pp. 1-4). Piscataway, NJ: IEEE Society Press.

Potdar, V., Sharif, A., & Chang, E. (2009). Wireless sensor networks: A survey. In *International Conference on Advanced Information Networking and Applications Workshops* (pp. 636-641). Piscataway, NJ: IEEE Society Press.

Reuther, B., & Henrici, D. (2008). A model for service-oriented communication systems, *Journal of Systems Architecture*, 54, 594–606.

Schaffers, H., *et al*. (2011). Smart cities and the future internet: towards cooperation frameworks for open innovation. In: Domingue, J., *et al*. (Eds.) *The future internet* (pp. 431-446). Berlin Heidelberg: Springer

Schippers, H. (2009). Towards an Actor-based Concurrent Machine Model, In Rogers, I. (Ed.)*4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (pp. 4-9).New York, NY: ACM Press.

Seinstra, F., *et al* (2011). Jungle computing: Distributed supercomputing beyond clusters, grids, and Clouds. In Cafaro, M. & Aloisio, G. (Eds.) *Grids, Clouds and Virtualization* (pp. 167-197). London, UK: Springer.

Severance, C. (2012). Discovering JavaScript Object Notation. *IEEE Computer*, 45(4), 6-8.

Sundmaeker, H., Guillemin, P., Friess, P., & Woelffle, S. (2010). Vision and challenges for realising the Internet of Things. *European Commission Information Society and Media*. Retrieved Sept. 12, 2013, from http://bookshop.europa.eu/en/vision-and-challenges-for-realising-the-internet-of-things-pbKK3110323/.

Upadhyaya, B., Zou, Y., Xiao, H., Ng, J., & Lau, A. (2011). Migration of SOAP-based services to RESTful services. In *13th IEEE International Symposium on Web Systems Evolution* (pp. 105-114). Piscataway, NJ: IEEE Society Press.

Wang, W., Tolk, A., & Wang, W. (2009). The levels of conceptual interoperability model: Applying systems engineering principles to M&S. In Wainer, G., Shaffer, C., McGraw, R. & Chinni, M. (Eds.), *Spring Simulation Multiconference* (article no.: 168). San Diego, CA:Society for Computer Simulation International.

Webber, J., Parastatidis, S., & Robinson, I. (2010). *REST in Practice: Hypermedia and Systems Architecture*. Sebastopol, CA: O'Reilly Media, Inc.

Weber-Jahnke, J., Peyton, L., & Topaloglou, T. (2012). eHealth system interoperability. *Information Systems Frontiers*, 14(1), 1-3.

Wyatt, E., Griendling, K., & Mavris, D. (2012). Addressing interoperability in military systems-of-systems architectures. In Beaulieu, A. (Ed.) *International Systems Conference* (pp. 1-8). Piscataway, NJ: IEEE Society Press.

Zikopoulos, P., *et al* (2012). *Understanding big data*. New York, NY: McGraw-Hill.

ADDITIONAL READING SECTION

Bell, M. (2008).*Service-Oriented Modeling: Service Analysis, Design, and Architecture*.New York, NY:John Wiley & Sons.

Chen, D., & Daclin, N. (2007). Barriers driven methodology for enterprise interoperability. In Camarinha-Matos, L., Afsarmanesh, H., Novais, P. & Analide, C. (Eds.) *Establishing The Foundation Of Collaborative Networks* (pp 453-460). Springer US

Chen, D., Doumeingts, G., & Vernadat, F. (2008). Architectures for enterprise integration and interoperability: Past, present and future. *Computers in Industry*, 59(7) 647–659.

Ford, T., Colombi, J., Graham, S., & Jacques, D. (2007). The interoperability score. In *Proceedings of the Fifth Annual Conference on Systems Engineering Research*, Hoboken, NJ.

Fricke, E., & Schulz, A. (2005). Design for changeability (DfC): Principles to enable changes in systems throughout their entire lifecycle. *Systems Engineering*, 8(4), 342-359.

Gehlert, A., Bramsiepe, N., & Pohl, K. (2008). Goal-Driven Alignment of Services and Business Requirements. In *International Workshop on Service-Oriented Computing Consequences for Engineering Requirements* (pp. 1-7). IEEE Computer Society Press.

Guédria, W., Chen, D., & Naudet, Y. (2009). A maturity model for enterprise interoperability. In Meersman, R., Herrero, P. and Dillon T. (Eds.) *On the Move to Meaningful Internet Systems Workshops* (pp. 216-225). Springer Berlin/Heidelberg.

Hoogervorst, J. (2004). Enterprise Architecture: Enabling Integration, Agility and Change.*International Journal of Cooperative Information Systems*, 13(3), 213–233.

Jardim-Goncalves, R., Grilo, A., Agostinho, C., Lampathaki, F., & Charalabidis, Y. (2013). Systematisation of Interoperability Body of Knowledge: the foundation for Enterprise Interoperability as a science. *Enterprise Information Systems*, 7(1), 7-32.

Jardim-Goncalves, R., Popplewell, K., & Grilo, A. (2012). Sustainable interoperability: The future of Internet based industrial enterprises. *Computers in Industry*, 63(8), 731-738.

Lewis, G., Morris, E., Simanta, S., & Wrage, L. (2008). Why Standards Are Not Enough To Guarantee End-to-End Interoperability. In Ncube, C. & Carvallo, J. (Eds.) *Seventh International Conference on Composition-Based Software Systems* (pp. 164-173). IEEE Computer Society Press

Luigi, A., Iera, A., & Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks*. 54(15), 2787–2805

Markov, I., & Kowalkiewicz, M. (2008). Linking Business Goals to Process Models in Semantic Business Process Modeling. In *International Enterprise Distributed Object Computing Conference* (pp. 332-338). IEEE Computer Society Press.

Ostadzadeh, S., & Fereidoon, S. (2011). An Architectural Framework for the Improvement of the Ultra-Large-Scale Systems Interoperability. In *International Conference on Software Engineering Research and Practice*. Las Vegas, NV.

Perepletchikov, M., Ryan, C., Frampton, K., & Tari, Z. (2007). Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. In *Australian Software Engineering Conference* (pp. 329-340). IEEE Computer Society Press.

Peristeras, V., & Tarabanis, K. (2006). The Connection, Communication, Consolidation, Collaboration Interoperability Framework (C4IF) For Information Systems Interoperability. *IBIS - International Journal of Interoperability in Business Information Systems*, 1(1) 61-72

Popplewell, K. (2011). Towards the definition of a science base for enterprise interoperability: a European perspective. *Journal of Systemics, Cybernetics, and Informatics*, 9(5), 6-11.

Quartel, D., Engelsman, W., Jonkers, H., & van Sinderen, M. (2009). A goal-oriented requirements modelling language for enterprise architecture. In *International conference on Enterprise Distributed Object Computing* (pp. 1-11).IEEE Computer Society Press.

Ross, A., Rhodes, D., & Hastings, D. (2008). Defining changeability: Reconciling flexibility, adaptability, scalability, modifiability, and robustness for maintaining system lifecycle value, *Systems Engineering*, 11 (3), 246-262.

Shroff, G. (2010).*Enterprise Cloud Computing: Technology, Architecture, Applications*. Cambridge, UK: Cambridge University Press.

Spohrer, J., Vargo, S., Caswell, N. & Maglio, P. (2008). The Service System is the Basic Abstraction of Service Science. In Sprague Jr., R. (Ed.) *41st Hawaii International Conference on System Sciences*. Big Island, Hawaii, 104, Washington, DC: IEEE Computer Society

Turnitsa, C. (2005). Extending the levels of conceptual interoperability model. In *IEEE Summer Computer Simulation Conference*. IEEE Computer Society Press.

Xu, X., Zhu, L., Kannengiesser, U., & Liu, Y. (2010). An Architectural Style for Process-Intensive Web Information Systems. In *Web Information Systems Engineering*, *Lecture Notes in Computer Science*, 6488 (pp. 534-547). Springer-Verlag Berlin Heidelberg.

Xu, X., Zhu, L., Liu, Y., & Staples, M. (2008). Resource-Oriented Architecture for Business Processes. In *Software Engineering Conference* (pp. 395-402), IEEE Computer Society Press.

KEY TERMS & DEFINITIONS

Resource: An entity of any nature (material, virtual, conceptual, noun, action, and so on) that embodies a meaningful, complete and discrete concept, makes sense by itself and can be distinguished from, although able to interact with, other entities.

Service: The set of operations supported by a resource and that together define its behavior (the set of reactions to messages that the resource exhibits).

Architectural style: A set of constraints on the concepts of an architecture and on their relationships.

Consumer: A role performed by a resource *A* in an interaction with another *B*, which involves making a request to *B* and typically waiting for a response.

Provider: A role performed by a resource *B* in an interaction with another *A*, which involves waiting for a request from *A*, honoring it and typically sending a response to *A*.

Compliance: An asymmetric property between a consumer *C* and a provider *P* (*C* is compliant with *P*) that indicates that *C* satisfies all the requirements of *P* in terms of accepting requests.

Conformance: An asymmetric property between a provider *P* and a consumer *C* (*P* conforms to *C*) that indicates that *P* fulfills all the expectations of *C* in terms of the effects caused by its requests.

Interoperability: An asymmetric property between a consumer *C* and a provider *P* (*C* is compatible with *P*) that holds if *C* is compliant with *P* and *P* is conformant to *C*.