Structural services– a new approach to enterprise integration

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

ABSTRACT

Currently, enterprise integration is based on SOA and/or REST, the two most used architectural styles for distributed interoperability. Web Services, the typical instantiation of SOA, constitute a complex technology with a sensible learning curve, even with development support tools. This has spurred a movement towards simpler solutions, namely those used in RESTful applications. This chapter analyzes the field of enterprise integration, comparing the SOA and REST architectural styles, and contends that both styles have advantages and limitations. SOA, based on behavior, has a lower modeling semantic gap for complex applications, but does not support structured resources, quite common at lower-grained applications. REST is based on structure and hypermedia, but has a higher semantic gap in complex applications and, as this chapter contends, does not entail a lower resource coupling than SOA. A new architectural style, Structural Services is proposed to get the best of both worlds while reducing coupling. Unlike REST, resources are able to offer a variable set of operations and, unlike SOA, services are allowed to have structure and use hypermedia. To reduce coupling, this style uses structural interoperability, based on the concepts of compliance and conformance. A distributed service programming language, SIL, is briefly described to illustrate how this architectural style can be instantiated.

INTRODUCTION

Enterprise integration is a very complex issue, spanning from lower levels, such as data and service interoperability, to the highest levels, including strategy and governance alignment. The latter are mostly dealt with tacitly or at the documentation level, in coordination with the enterprise architectures. The former typically resort to technologies based on architectural styles such as SOA and REST and constitute the focus of this chapter, although the others are tackled as well, in less detail.

SOA is guided by behavior and REST by (representation of) state. In UML terminology, SOA uses a static class diagram 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, 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 the original problem (integrating enterprises) is the same.

REST is 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. This however, 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. This is why REST favors standardized media types, to reduce the coupling between the client and the server. However, instead of reducing it, it merely transfers it from runtime to design time, when media types are defined. SOA is more static, since the service contracts are defined beforehand and there is no provision for resource structure, but benefits from the support provided by tools in detecting errors. This is the classical tradeoff that a programmer has to face when choosing between a dynamic or static typed language.

In REST, there is an interesting side effect of the fact that the client uses a response to perform the next state transition. Only the links corresponding to allowed transitions are there, in line with the *affordances* approach (Amundsen, 2012). This is equivalent to dynamically adjusting the interface available to the client, according to the current state of the application. This is good protection mechanism, already used in human computer interfaces, with menus that become enabled and disabled, according to context. In SOA, this is not easy to achieve. Once a link to a service is available, all its operations can be invoked.
The flexibility of REST comes at a price, since it lies at a much lower level than SOA and the architecture of the problem does not map as easily onto state diagrams as onto resource class diagrams and processes. Nevertheless, it is simpler than SOA, requiring only basic Web technologies, and scales much better, but only as long as the application complies with the constraints imposed by REST. Complex functionality and bidirectional interactions (requests from server to client) are not a good match for REST.
This means that the ranges of types of applications that are more adequate for SOA and REST are not the same. REST is a better match for Web style applications (many clients, stateless interactions), which provide a simpler interface. This is why all the major Internet application providers, including cloud computing, now have REST interfaces. SOA is a better match for functionally-complex, enterprise-level distributed applications, in which REST exhibits a higher semantic gap (Ehrig, 2007) between the problem and solution spaces.
Ideally, we would like:

- To combine the structural properties of REST's resources and links with the richness of generic service interfaces;
- To reduce the semantic gap of REST, without losing its dynamic structural characteristics;
- To reduce the coupling between the provider and consumer of a service (increasing its range of applicability and adaptability), without losing its contract-based support.

Unfortunately, the technologies currently used to implement SOA and REST (XML, JSON, HTTP, SOAP, WSDL, and so on) provide no hint or support on how to do this.
This is a direct consequence of the fact that these technologies evolved from the original Web of Documents, made for stateless browsing and with text as the dominant media type. That web was made for people, not for enterprise integration. Today, the world is rather different. The web is now the Web of Services, in which the goal is to provide functionality, not just documents. There are now more computers connected than people, with binary data formats (computer data, images, video, and so on) as the norm rather than the exception.
Yet, the evolution has been to map the abstraction of Web of Services onto the Web of Documents, with the major revolution of XML and its schemas. The document abstraction has been retained, with everything built on top of it. The lack of support of XML for behavior (service paradigm) is one of the main sources of complexity.
This chapter adopts a model-driven approach, rather than technology-driven, in the sense that it follows the typical system lifecycle, progressing from the vision of what an enterprise and its integration with others should be down to its implementation, instead of starting with the current technologies and see how these can be combined to achieve enterprise integration.
The main goals of this chapter are:

- To contribute to the field of enterprise integration, by showing that revisiting the interoperability problem can yield an alternative solution that is both simpler and more effective than existing ones;
- To describe a framework that explores the various dimensions of enterprise integration, to better dissect and understand the problem;
- To propose a new architectural style that combines the best characteristics of SOA and REST, endowing resources with behavioral, structural and hypermedia capabilities;

- To introduce a new interoperability language, which considers resources and their services natively, contemplating both data and behavior, instead of building everything on top of a data description language;
- To assess the potential of conformance or compliance as a means to reduce coupling and increase adaptability and changeability, while maintaining interoperability requirements.

The chapter is organized as follows. The Background section describes some of the existing technologies relevant to the context of this chapter. Next, the integration problem is described in greater detail, followed by a discussion of integration paradigms and by the presentation of the integration model used in this chapter, which also describes two frameworks, one to derive an enterprise architecture and the other to systematize the interoperability aspects involved in the interaction between enterprise architectures. A fundamental issue is to reduce coupling, for which this chapter proposes compliance and conformance as the main underlying concepts. Next, a hierarchy of integration architectural styles is presented, including a comparison between the main characteristics of SOA and REST. A new architectural style, Structural Services, is proposed and compared with SOA and REST. A programming language, SIL, is briefly illustrated to show how this style can be used in practice. The chapter ends by outlining future directions of research and by drawing the main conclusions of this work.

BACKGROUND

Interoperability, a necessary condition to achieve integration, has been studied in domains such as enterprise cooperation (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).
The original Web of Documents has evolved into a global Web of Services, in which the interacting parties are both humans and computer applications, while content is increasingly dynamic, generated on the fly according to some database and business logic. The underlying logical architecture became less client-server and more peer based, in particular when enterprise applications are involved.
SOA (Earl, 2008), instantiated by Web Services, embodied a major evolution, the transition from the client-server to the service paradigm. Web services required a new protocol (SOAP) and a way to express the interfaces of services (WSDL). HTTP and HTML are technologies that were originally conceived for large-scale applications and human clients. XML, which maintained the text markup style of HTML, made computer based clients easier and, together with HTTP, became the cornerstone of Web Services technologies. This evolutionary transition is perfectly understandable in market and standardization terms, but still constitutes a mismatch towards both humans and applications.
HTTP is an application level protocol, synchronous and committed to the client-server paradigm. 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). Web Sockets, now part of the HTML5 world (Lubbers, Albers & Salim, 2010), removes this restriction, adds binary support and increases performance.
XML is verbose and complex, has limited support for binary formats, is inefficient in computer terms due to parsing and exhibits symmetric interoperability, based on both sender and receiver using the same schema, which constitutes a relevant coupling problem.
The notion of schema matching (Jeong, Lee, Cho & Lee, 2008) is used only for service discovery, not for message exchange. This means that Web Services are more a legacy integration and interoperability mechanism than a true and native service oriented solution. Their universal service interoperability came at the price of complexity, with schemas and all the associated standards. Tools automate and simplify a lot, but they just hide the complexity and do not eliminate it. This has spurred a movement towards simpler, more manageable systems, in the form of a resource oriented architectural style, REST (Fielding,

2000), and of a simpler data format, JSON (Galiegue & Zyp, 2013), promoting what is known as RESTful applications (Li and Chou, 2010).
Whereas SOA emphasizes behavior and rich interfaces, the guiding aspects of REST are state, structure and uniform interfaces. All resources implement the same service, with the same set of operations, albeit with different implementations. These typically exhibit CRUD style (Create, Read, Update and Delete) and are mapped onto HTTP verbs (PUT, GET, POST and DELETE). Functionality that in SOA would be modeled as an ~~operation~~ is modeled in REST as a ~~resource~~.
REST is defined at a lower level than SOA, with a higher semantic gap between the problem and solution spaces. However, it maps directly onto the familiar HTTP protocol and has some interesting scalability properties, which are important for the class of applications in which a server is used by many clients.
In fact, armored with simplicity and scalability, the REST style has become very popular (Pautasso, Zimmermann & Leymann, 2008), not only among services in the web but also in cloud computing. All the major providers now offer a REST API to their systems.
The scientific community has engaged in a continuing debate over which architectural style, SOA or REST, is more adequate for specific classes of applications. The literature comparing these styles is vast (Mulligan & Gracanin, 2009; Becker, Matzner & Müller, 2010), usually with arguments more on technology issues than on conceptual and modeling arguments. The REST proponents seem to be more active, since this style was rediscovered (or at least became more active) more recently than the standardization of Web Services and their adoption by the market. Besides arguing why REST is better (Pautasso, Zimmermann & Leymann, 2008), a recurring theme has been to try to demonstrate that REST is also adequate for the enterprise class of applications (Pautasso, 2009).
Peng, Ma and Lee (2009) proposed a framework to integrate SOA and RESTful services, allowing them to coexist and interoperate.

THE INTEGRATION PROBLEM

We start by informally laying down some definitions to express the context in which we understand the concept of integration:

- **Resource**: An entity of any nature (material, virtual, conceptual, noun, action, and so on) that embodies a meaningful, complete and discrete concept, making sense by itself while being distinguishable from, and able to interact with, other entities. A resource can be atomic (an indivisible whole, or a black box) or structured (composed of other resources, recursively);
- **System**: A structured resource (a set of interrelated and interacting resources);
- **Model**: An abridged representation of a given system and the relationships between its resources, in the form of a simplified specification of that set (abstract view of the problem, stemming from analysis);
- **Architecture**: A specific organization of a system, providing a concretization of a model in the form of a simplified implementation of that system (abstract view of the solution, stemming from design);
- **Paradigm**: A set of concepts and patterns that guide the construction of the model and the design of the architecture;
- **Architectural style**: A collection of design patterns, guidelines and best practices (Dillon, Wu & Chang, 2007), or as a set of constraints on the concepts and on their relationships (Fielding & Taylor, 2002), usually under a given paradigm;
- **Framework**: A set of principles, assumptions, rules and guidelines to analyze, to structure and to classify the concepts and concerns of a system or its domain. A given paradigm may be assumed;
- **Method**: A set of techniques, best practices and guidelines to yield a solution to a problem (in a given system), by building a model and deriving an architecture, in the context of a framework;

- **Integration**: The act of instantiating a given method to design or to adapt two or more resources, so that they are able to cooperate and to accomplish one or more common goals;
- **Interoperability**: The ability of two or more resources to meaningfully exchange information to operate together (ISO/IEC/IEEE, 2010). As an ability, interoperability is a necessary but not sufficient condition for integration, which must realize the potential provided by interoperability through the correct use of the method chosen;
- **Lifecycle**: The set of stages that a resource goes through during its lifetime, from initial conception to termination, when no longer in use;
- **Distributed system**: A system in which the lifecycles of resources are not synchronized (may evolve independently). Distribution does not necessarily entail geographical dispersion;
- **Coupling**: A value between 0 and 1 that expresses the degree of dependency between two resources. A value of 0 means that a resource does not depend on (or cannot be affected by) any of the characteristics of the other resource. A value of 1 means that a resource depends on all (or may be affected by any of) the characteristics of the other resource. Intermediate values express partial dependencies;
- **Decoupling**: The opposite notion of coupling, or its complement to 1. A value of 0 means complete dependency and a value of 1 means complete independency.

In the general, the integration problem revolves around two conflicting goals:

- **Interoperability**: Resources need to interact to accomplish collaboration, either designed or emergent. This necessarily entails some form of mutual knowledge and understanding, but this creates dependencies on other resources that may hamper resource evolution (a new iteration in the lifecycle);
- **Decoupling**: Resources need to be independent to evolve freely and dynamically. Unfortunately, independent resources do not understand each other and are not able to interact.

Therefore, the *fundamental problem of resource integration* is to provide the *maximum decoupling* possible while ensuring the *minimum interoperability* requirements.
In other words, the main goal is to ensure that each resource knows enough to be able to interoperate but no more than that, to avoid unnecessary dependencies and constraints. This is an instance of the *principle of least knowledge*, also known as the Law of Demeter (Palm, Anderson & Lieberherr, 2003).
Enterprises are complex systems and suffer exactly from this problem, with the additional concern that agility (which translates to fast evolution) is a critical requirement for the survival of even the largest enterprises. These are in fact quite vulnerable, because their complex architecture has typically woven a large web of dependencies. Current integration technologies do not necessarily comply with the principle of least knowledge.
As mentioned in the Background section, interoperability is an extensively researched topic. Decoupling is a much less studied issue, since interoperability is simply based on sharing schema files, typically in the world of Web Services, or predefined media types, as used by RESTful applications. In either case, data types need to be fully known by both interacting resources. In this chapter, we show how interaction is still possible with only a partial knowledge of types, as long as the characteristics actually used are included (partial interoperability). This is a way of getting closer to solving the fundamental integration problem, by reducing coupling to what is actually required.
The following sections detail the most relevant integration aspects that were defined above.

CHOOSING AN INTEGRATION PARADIGM

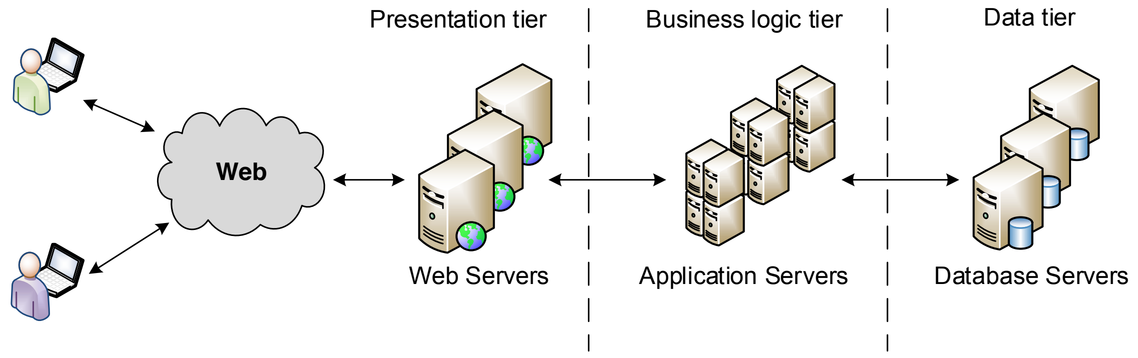
To study integration, we need models, architectures, frameworks, and so on. This implies settling on a resource interaction paradigm, on which these artifacts can be based. Two of the most important and

common paradigms in distributed resource interaction are processes and services. Choosing one or the other does not imply that processes and services are mutually exclusive. In fact, all active systems have both. Processes cannot exist without services (the units of behavior that processes invoke) and services originate processes when the chain of service to service invocations is unraveled at any service's entry point. The relevant issue is which concept is at the top, process (activity perspective) or service (architectural perspective).
For example, the Zachman framework separates data (what) from function (how). TOGAF separates the data architecture from the application architecture. Other enterprise architecture frameworks (Minoli, 2008) do essentially the same. They are recursive, in the sense that they can be applied to a large enterprise as well as to a small module in an application. In practice, however, granularity tends to be very high. The complexity of the entire set of considerations and rationale is enough to grant its applicability only to the entire enterprise or to reasonably sized units within an enterprise.
In this context, and by separating data from applications, these frameworks implicitly assume that:

- There is a shared data repository, with the existence of a canonical data model as a typical architectural principle (Greefhorst & Proper, 2011);
- Applications read, process and update the data stored in this shared repository.

In practice, there is an implicit choice of the process paradigm that is inherent to the frameworks themselves. This separation is a good match to the N-tier architectural style, illustrated by Figure 1, in which the application logic (processes) is separated from the database. This is not merely a question of paradigm, but also a result of technological issues (such as database implementation and optimization). The presentation tier is also separated, but for different reasons (such as interface independence and security).

Figure 1. Example of a typical N-tier system.



This centralization of data and its separation from applications is a classical, but unnecessary assumption and restriction that can raise many problems in enterprise integration, when business processes cross the boundaries of the architecture of an enterprise. Each tier is, in practice, a functional silo.
The same can be said about the maintainability and adaptability of an enterprise's architecture, in which unintended effects of changes easily propagate throughout the shared data. It is hard to keep track of which data is used by or affects which processes and CRUD matrices do not scale well with complexity.
Another disadvantage of this paradigm is that the architecture development method has to deal with the semantic gap between the business, seen as a set of interacting entities, both inter- and intra-enterprise, and the data/application architecture level, which removes the boundaries of these entities instead of modeling them as they appear at the business level.
Object-oriented programming has long abandoned the shared data model of structured programming, by structuring the programs according to some principles that tend to limit the impact of a change (affecting

fewer objects) and to increase the adaptability (allowing objects to be adapted, even without the source code), namely:

- **Least semantic gap principle**: The entities in the problem and solution spaces should have a one to one mapping, by modeling each problem entity with a solution object, with includes both state and related behavior. Any model will be easier to change if its abstractions match the entities it models. A sale, for example, will be the result of the cooperation of several entities (users, employees, information system, logistics, etc.). These entities are much closer to the service paradigm, since processes do not exist as real world entities. A small semantic gap (Ehrig, 2007) reduces the maintainability and adaptability efforts. A small change in the system requirements translates to a small change in the system's architecture, due to that close match;
- **Information hiding principle**: There must be a clear separation between interface (syntax and semantics) and its implementation (both code and data). This reduces the change propagation effect. A change in one object affects those that use it only if its interface is affected. Processes are usually modeled after use cases (global uses of a system) and therefore tend to span across a large part of that system. Several processes will use the same service. A change in one service has potential impact in many processes. In contrast, services use an outsourcing mechanism (when invoking other services), but hide it from the services that have invoked them.

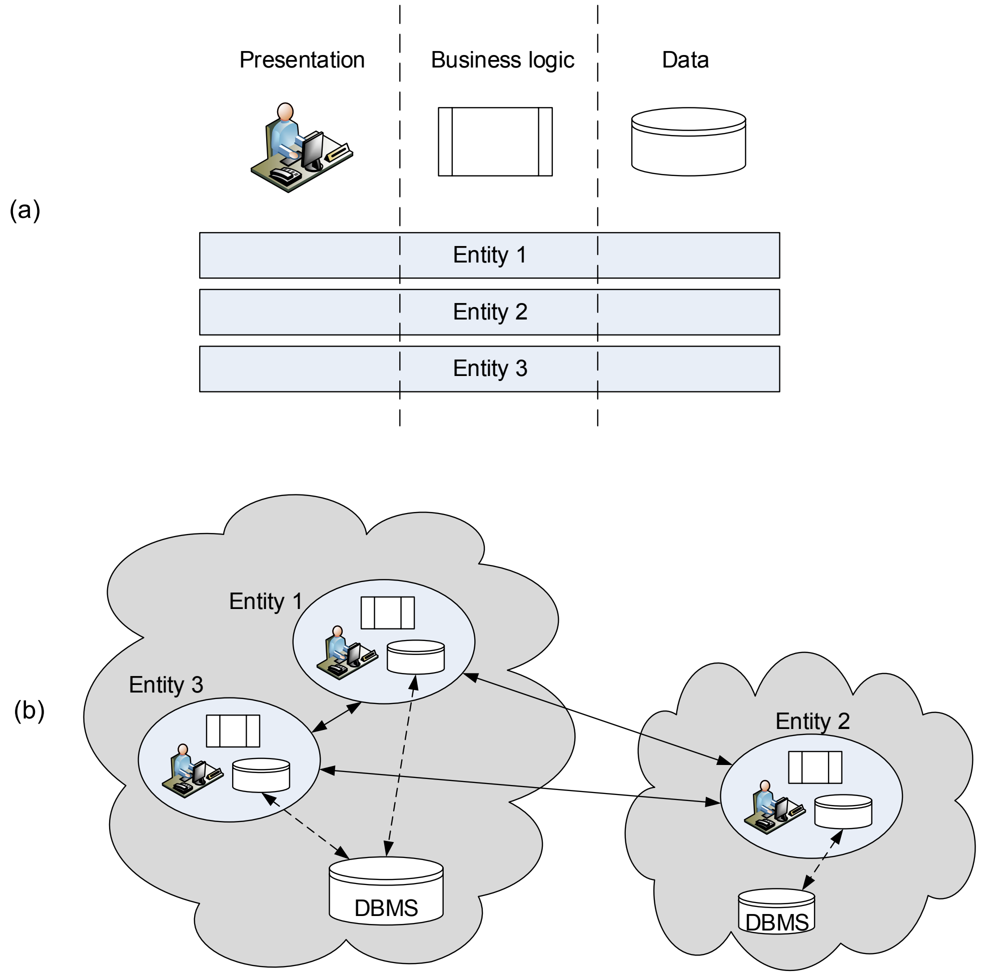
Resources correspond to distributed objects, but the same basic principles should apply. The goal is to benefit from the advantages heralded by object-oriented programming in the enterprise architecture context, without being restricted to a centralized application.
The relationship between resources, services and processes can be described in the following way:

- A *service* is a set of logically related operations of a resource. In other words, it is a facet of that resource that makes sense in terms of modeling of the envisaged resource. A service is pure behavior, albeit the implementation of concrete operations may depend on state, which needs a resource as an implementation platform;
- Resources interact by sending each other *messages*. A resource *A* that sends a message to a resource *B* is in fact invoking the service of *B*. This constitutes a *service transaction* between these two resources, in which the resources *A* and *B* perform the role of service *consumer* and service *provider*, respectively. A service transaction can entail other service transactions, as part of the chain reaction to the message on part of the service provider. A service should be defined in terms of reactions to messages (external view) and not in terms of state transitions or activity flows (internal view);
- A *process* is a graph of all service transactions that are allowed to occur, starting with a service transaction initiated at some resource *A* and ending with a final service transaction, which neither reacts back nor initiates new service transactions. The process corresponds to a use case of resource *A* and usually involves other resources as service transactions flow.

Resources entail structure, state and behavior. Services refer only to behavior, without imposing a specific implementation. Processes are a view on the behavior sequencing and flow along services, which are implemented by resources. Services and processes are dual of each other, having resources as the structural entities. Where there is a service, there is a process and vice-versa. The question is which paradigm should be used first and foremost in modeling an enterprise's architecture, with the other derived from it.
We contend that, in terms of modeling, adaptability and changeability, the service paradigm is much better than the process paradigm, and a decisive contribution to the agility of enterprises (Uram & Stephenson, 2005).

Figure 2 illustrates this. In a classical N-tier architecture, each entity is spread throughout the various tiers, as depicted in Figure 2a. By changing to a service-oriented perspective, we group all the pieces of a given entity in each tier into one resource, which becomes a first class citizen in a tierless graph of interacting resources. Each of these resources includes (conceptually) its own independent database, holding the permanent data. In terms of implementation, the resources that reside in the same server can map their conceptual databases onto the same DBMS, but with logically separate records (Ahmad & Bowman, 2011), as depicted in Figure 2b, which can be tables or objects, depending on the underlying model of the DBMS. Note that this has no impact on the service provided by each record, since the service is an interface at the entity level, not at the database access level.
This is a flexible arrangement (since data is logically partitioned and bound with the behavior that manages that data), which exploits the essential advantages of services referred to above in a distributed environment, either between enterprises or within the enterprise itself. The object-oriented paradigm has evolved into the service-oriented paradigm.

Figure 2. Migration from a centralized N-tier system (a) to a distributed tierless system (b).

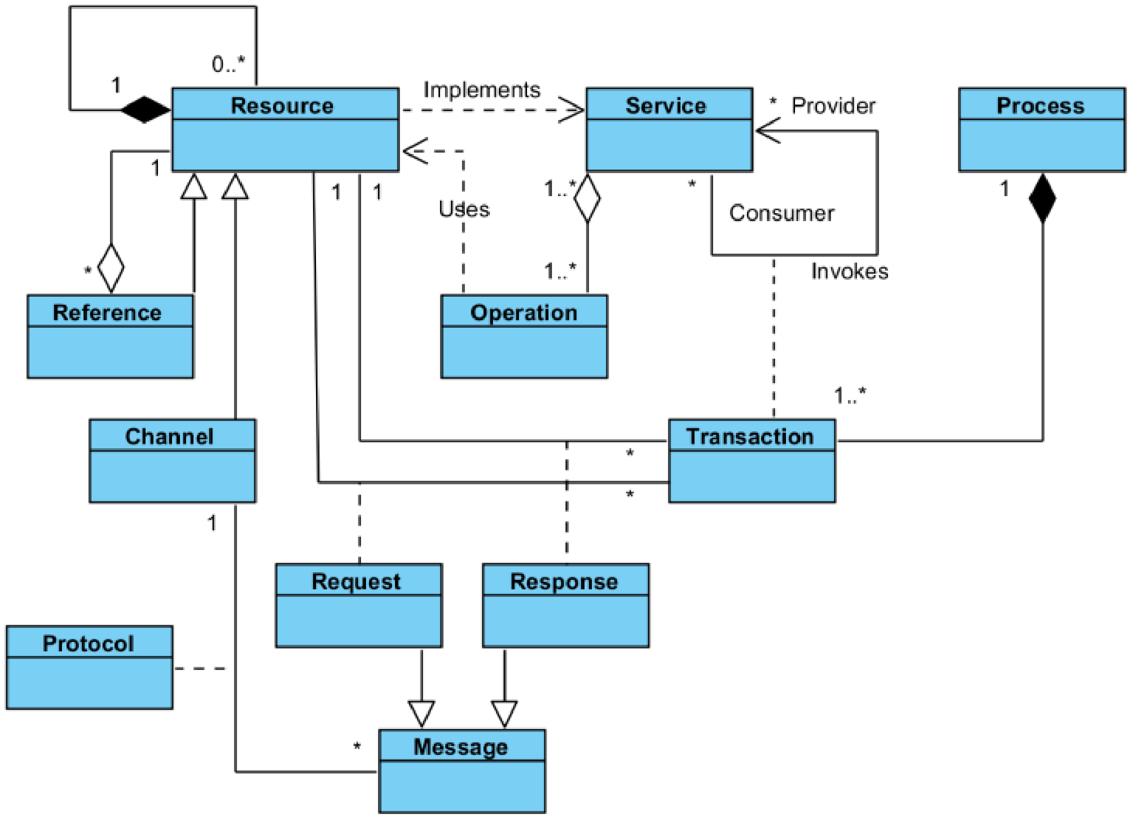


AN INTEGRATION MODEL

Resources can be simple or complex. We need an abstract model to express the relationships between the entities described above, such as resources, services, processes, messages and transactions. The goal is to have a model that can deal with enterprise artifacts and their integration in a uniform way.

Figure 3 depicts such a model. A resource provides implementation for a service, namely its operations. Services invoke each other through service transactions, forming a process. Interaction is message based, assuming that an adequate channel and message protocol are available. Resources can be composed of other resources (composition) or have references to other resources (aggregation). References themselves are resources and included as components (but not the resources they reference) in their containers.

Figure 3. A simple model of system interaction.



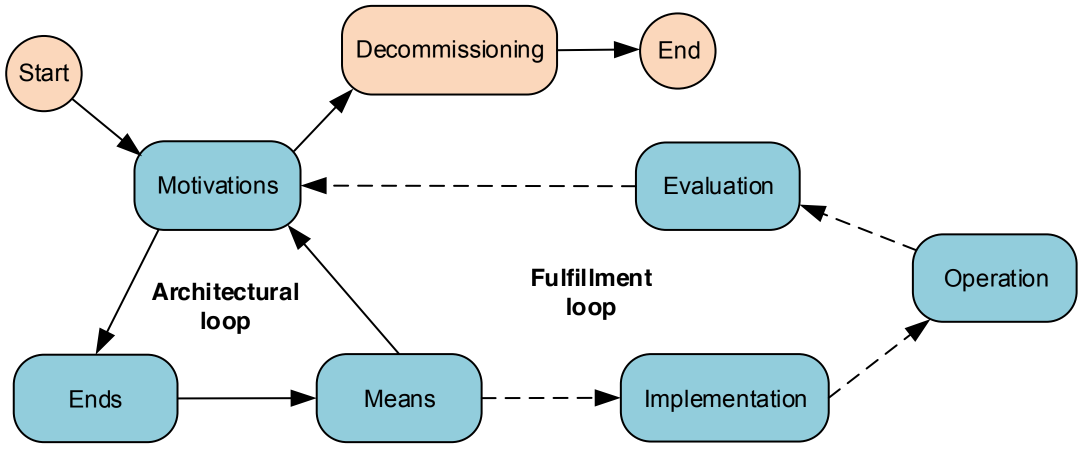
Association classes describe relationships (such as Transaction and the message Protocol) or roles, such as Message, which describes the roles (specialized by Request and Response) performed by resources when they are sent as messages.
The interaction between resources has been represented as interaction 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.

AN ENTERPRISE ARCHITECTURE FRAMEWORK

Integration can be used in the context of enterprises to adapt existing applications or to help designing enterprise architectures, which should be conceived for integration right from the start. We are more interested in the latter case, since adapting existing applications, although a real and fundamental need, is heavily conditioned by existing specifications and does not allow us to use the full range of integration aspects and possibilities.
However, before designing for integration, we need to know how to design resources. These can range from the complete enterprise architecture (representing the very complex resource that the enterprise constitutes) down to very simple modules (depending on the resource granularity considered).

One of the first things to realize is that integration does not just occur when the whole system is operating. It starts much earlier, in the conception of resources. Integration must be considered in the full resource lifecycle. Figure 4 depicts a simplified view of a typical lifecycle, which includes several stages that reflect different perspectives, from initial conception to final decommissioning. The loops support iteration of the lifecycle, to cater for changes and improvements.

Figure 4. Resource lifecycle, with stages chained in changeability loops.



The *architectural loop* is inspired by the Business Motivation Model (Malik, 2009) and contemplates three main concepts, which embody three of the main questions about system development that were popularized by the Zachman framework (O'Rourke, Fishman & Selkow, 2003), *why* (motivations), *what* (ends) and *how* (means):

- **Motivations**: Emphasize the reasons behind the architectural decisions taken, in accordance with the specification of the problem that the resource is designed to solve;
- **Ends**: Express the desires and expectations (i.e., goals and objectives) of the stakeholders for which the resource is relevant;
- **Means**: Describe the mechanisms used to fulfill those expectations (i.e., actions).

This loop should be iterated until a satisfactory specification exist for the outcome of each stage.
The *fulfillment loop* in Figure 4 models subsequent stages and is based on the organization adopted by classical development methods, such as the Rational Unified Process (Kruchten, 2004):

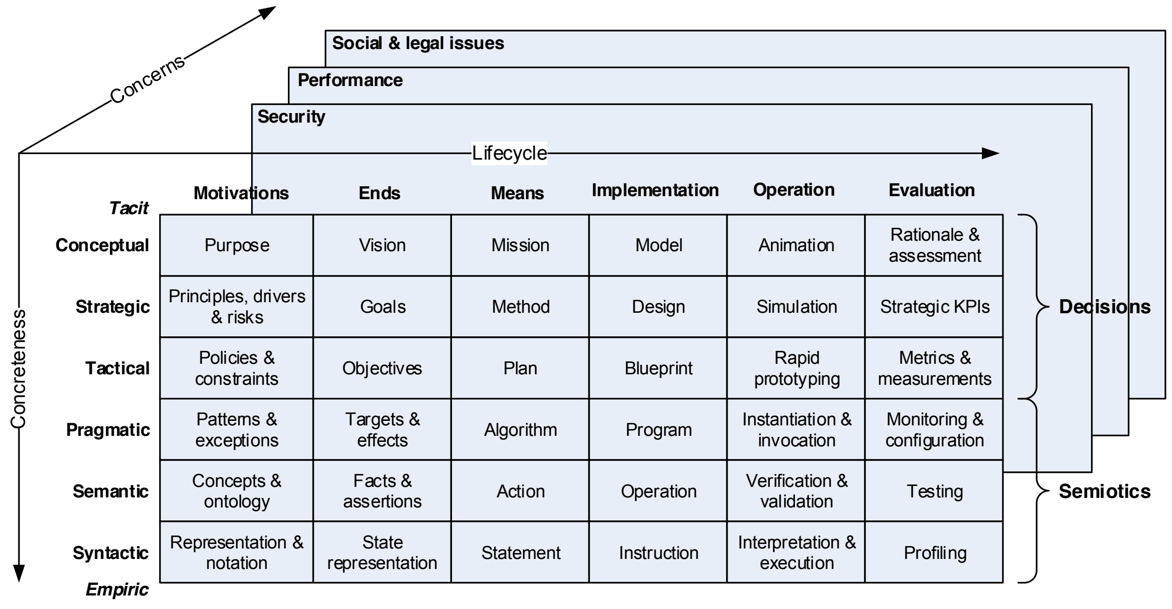
- The Ends and the Means of the architectural loop correspond roughly to the analysis and design stages of those methods;
- **Implementation**: Includes stages such as development, testing and deployment;
- **Operation**: Corresponds to executing the system;
- **Evaluation**: Measures key performance indicators (KPIs) and assesses how well the system meets the expectations stemming from the motivations.

Iteration of the fulfillment loop goes back to Motivations, so that the resource can be reconsidered again from the origin of the lifecycle. If the motivations are no longer considered worthwhile, the lifecycle ends with the decommissioning of the resource.
The lifecycle is just one dimension (or axis) of our architectural framework, which includes the following dimensions (or axes):

- **Lifecycle**: This axis is discretized into the six stages of Figure 4 (ignoring the Decommissioning stage, less relevant to integration design);
- **Concreteness**: Each stage in the lifecycle (Figure 4) can be viewed at a very high, abstract level, or at a very detailed, concrete level. We have discretized this axis into six levels, depicted in Figure 5: *Conceptual*, *Strategic*, *Tactical*, *Pragmatic*, *Semantic* and *Syntactic*. Stages of the lifecycle and their level of concreteness are orthogonal concepts. We can have the Implementation stage at a conceptual level (just ideas on how to do it) as well as the Motivations stage at a very concrete level (justification for the lowest level actions);
- **Concerns**: The focus words (*what*, *how*, *where*, *who*, *when*, *why*) in the Zachman framework (O'Rourke, Fishman & Selkow, 2003) are generic but do not address the entire focus range. Other questions are pertinent, such as *whence* (where from), *whither* (where to), *how much* (quantitative assessment) and *how well* (qualitative assessment). It is important to be able to express the dynamics of the architecture of the resource, its quality (how good it is, quantitatively and qualitatively) and other concerns (performance, standards, security, reliability, and so on), both functional and non-functional. This axis is discretized as needed, according to the number of concerns considered.

Figure 5 illustrates this framework and its three axes. The front plane is formed by the Lifecycle and Concreteness axes. Each cell, resulting from crossing the values of both axes, represents one lifecycle stage at a given level of concreteness. Each row is a refinement of the level above it, by including decisions that turn some abstract aspects into concrete ones. Each concern, functional or non-functional, represents a new plane, along the Concerns axis. We have illustrated some concerns, with plane details omitted for simplicity.

Figure 5. The axes of the architecture framework, with the front plane detailed.



	Motivations	Ends	Means	Implementation	Operation	Evaluation
Conceptual	Purpose	Vision	Mission	Model	Animation	Rationale & assessment
Strategic	Principles, drivers & risks	Goals	Method	Design	Simulation	Strategic KPIs
Tactical	Policies & constraints	Objectives	Plan	Blueprint	Rapid prototyping	Metrics & measurements
Pragmatic	Patterns & exceptions	Targets & effects	Algorithm	Program	Instantiation & invocation	Monitoring & configuration
Semantic	Concepts & ontology	Facts & assertions	Action	Operation	Verification & validation	Testing
Syntactic	Representation & notation	State representation	Statement	Instruction	Interpretation & execution	Profiling

We consider the levels in the Concreteness axis organized in two categories:

- **Decisions** taken, which define, structure and refine the characteristics of the resource, at three levels:
  - **Conceptual**: The top view of the resource, including only global ideas;
  - **Strategic**: Details these ideas by taking usually long lasting decisions;
  - **Tactical**: Refines these decisions into shorter-term decisions;

- **Semiotics** (Chandler, 2007): The study of the relationship between signs (manifestations of concepts) and their interpretation (pragmatics), meaning (semantics) and representation (syntax). In this chapter, these designations correspond to the following levels:
  - **Pragmatic**: Expresses the outcome of using the resource, most likely producing some effects, which will depend on the context in which the resource is used;
  - **Semantic**: Specifies the meaning of the resource, using an ontology to describe the underlying concepts;
  - **Syntactic**. Deals with the representation of the resource, using some appropriate notation or programming language.

All the concreteness levels express a range between two opposite thresholds, also represented in Figure 5:

- **Tacit**: This is the highest level, above which concepts are too complex or too difficult to describe. It encompasses the tacit knowledge and know-how (Oguz & Sengün, 2011) of the people that designed the resource, expressing their insight and implicit expectations and assumptions about the problem domain;
- **Empiric**: The lowest level, below which details are not relevant anymore. The resource designers just settle for something that already exists and is known to work, such as a standard or a software library.

A method will be needed to exercise the architecture framework, to navigate from the top-left cell in Figure 5 (conceptual motivations, or purpose) to the bottom-right cell (operation at the syntactic level, or interpretation & execution), with the rightmost column taking care of evaluation to decide whether to loop back for architectural evolution. The path taken determines the method to use. The usual approach is to follow the diagonal, more or less, reflecting the fact that advancing in the lifecycle stages also provides further detail because decisions are taken along the way, increasing the concreteness level of the design. But other approaches are also possible, more breadth-first or more depth-first. This must be done in all planes and deciding which planes to tackle first also leads to method variants. This methodology is outside the scope of this chapter.

AN INTEROPERABILITY FRAMEWORK

Now that we have gained some insight into the design of a resource, we need to understand what is involved when that resource *A* needs to interact with some other resource *B*. Suppose that *A* sends a message to *B*. The definition of interoperability, given in section "The integration problem", requires a meaningful exchange of information, which translates to requiring that:

- The message reaches its destination, *B*, after traversing the channel (Figure 3);
- The message it is correctly read and its meaning well understood by *B*, both in terms of message content and of the purpose of *A* in sending the message;
- The reaction of *B* to the message honors the purpose of *A*.

If this message is a request from a consumer to a provider, the same can be said about the response, with the roles of sender and receiver reversed, thereby completing a service transaction (Figure 3).
Most integration technologies, such as Web Services and RESTful APIs, consider only the syntactic format of messages, or at most the semantics of its terms. As Figure 5 shows, the ability to meaningfully interact involves higher levels (upper levels in the Concreteness dimension) and lifecycle stages prior to (to the left of) Operation. If the Concerns dimension is taken into account, things get even more complex. For example, an interaction may not work due to response times too long (making timeouts expire), even if functionally the design is correct.

To understand the interoperability aspect completely, we need an interoperability framework. The basic tenet is that it should be well matched to the architecture framework. In particular, the resource *A* should be able to interact with resource *B* if each cell of the three-dimensional space (Figure 5) of the architecture of resource *A* satisfies the requirements of the corresponding cell in the three-dimensional space of the architecture of resource *B*. This is valid even in the design stages, before Operation. For example, the motivations of *A* in sending a message should match the motivations of *B* to receive it and to honor its purpose.
In practice, most of these interactions between corresponding cells are dealt with *tacitly*, not explicitly. In particular, stages prior to Operation are usually tackled at the documentation level. Nevertheless, they are very relevant, because what happens in the design stages allows us to understand the true capabilities and limitations of integration solutions and technologies, which is the main purpose of this chapter. Nevertheless, the Operation stage is the most visible, in terms of interoperability. Table 1 establishes a classification of interoperability in layers, with a rough correspondence to the values in the Concreteness axis in Figure 5.

Table 1. Layers of interoperability in a transaction.

Category	Layer	Main artifact	Description
Symbiotic (purpose and intent)	Coordination Alignment Collaboration Cooperation	Governance Joint-venture Partnership Outsourcing	Motivations to have the transaction, with varying levels of mutual knowledge of governance, strategy and goals
Pragmatic (reaction and effects)	Contract Workflow Interface	Choreography Process Service	Management of the effects of the transaction at the levels of choreography, process and service
Semantic (meaning of content)	Inference Knowledge Ontology	Rule Semantic network Concept	Interpretation of a message in context, at the levels of rule, relations and definition of concepts
Syntactic (notation of representation)	Structure Predefined type Serialization	Schema Primitive resource Message format	Representation of resources, in terms of composition, primitive resources and their serialization format in messages
Connective (transfer protocol)	Messaging Routing Communication Physics	Message protocol Gateway Network protocol Equipment	Lower level formats and network protocols involved in transferring a message from the context of the sender to that of the receiver

The first column in Table 1, as well as its subdivisions in the second column, should be interpreted as follows:

- **Symbiotic**: Expresses the purpose and intent of two interacting resources to engage in a mutually beneficial agreement This can entail a tight coordination under a common governance (if the resources are controlled by the same entity), a joint-venture agreement (if the two resources are substantially aligned), a collaboration involving a partnership agreement (if some goals are shared), or a mere value chain cooperation (an outsourcing contract). Enterprise engineering is usually the topmost level in resource interaction complexity, since it goes up to the human level, with governance and strategy heavily involved. Therefore, it maps mainly onto the symbiotic layer, although the same principles apply (in a more rudimentary fashion) to simpler resources;
- **Pragmatic**: The effect of an interaction between a consumer and a provider is the outcome of a contract, which is implemented by a choreography that coordinates processes, which in turn implement workflow behavior by orchestrating service invocations. Languages such as BPEL

support the implementation of processes and WS-CDL is an example of a language that allows choreographies to be specified;

- **Semantic**: Both interacting resources must be able to understand the meaning of the content of the messages exchanged, both requests and responses. This implies interoperability in rules, knowledge and ontologies, so that meaning is not lost when transferring a message from the context of the sender to that of the receiver. Semantic languages and specifications, such as OWL and RDF, map onto this category;
- **Syntactic**: Deals mainly with form, rather than content. Each message has a structure, composed of data (primitive resources) according to some structural definition (its schema). Data need to be serialized to be sent over the channel as messages, using formats such as XML or JSON;
- **Connective**: The main objective is to transfer a message from the context of one resource to the other's, regardless of its content. This usually involves enclosing that content in another message with control information and implementing a message protocol (such as SOAP or HTTP) over a communications network, according to its own protocol (such as TCP/IP) and possibly involving routing gateways.

It is very important to recognize that all these layers are always present in *all* resource interactions, even the simplest ones. There is always a motivation and purpose in sending a message, an effect stemming from the reaction to its reception, a meaning expressed by the message and a format used to send it over a channel under some protocol.
However, what happens in practice is that some of these layers are catered for *tacitly* or *empirically*. The Connective layer is usually dealt with empirically, by assuming HTTP or SOAP. The Syntactic layer is the main workhorse of integration technologies, with data description languages such as XML and JSON. The Semantic layer, giving the increasing relevance of the Semantic Web (Shadbolt, Hall & Berners-Lee, 2006), is beginning to be explicitly addressed with semantic annotations and languages, based either on XML or JSON, but it is still largely subject to tacit assumptions. XML's namespaces provide precious help here. The Pragmatic layer is usually considered at the documentation level only, with many tacit assumptions. Choreography specifications and tools to verify them are rarely used. The Symbiotic layer is considered only, and again at the documentation level, with very complex resources (such as enterprises) and at the conceptual stages of the lifecycle. In most cases, it is simply assumed that, if a resource exposes its service, it can be invoked regardless of motivations, purpose or any other high-level concerns.
Another important aspect is non-functional interoperability. It is not just a question of invoking the right operation with the right parameters. Adequate service levels, context awareness, security and other non-functional issues must be considered when resources interact, otherwise interoperability will be less effective or not possible at all. The framework of Table 1 must be considered for each plane in Figure 5.
Finally, we must realize that all these interoperability layers constitute an expression of resource coupling. On one hand, two uncoupled resources (with no interactions between them) can evolve freely and independently, which favors adaptability, changeability and even reliability (if one fails, there is no impact on the other). On the other hand, resources need to interact to cooperate towards common or complementary goals, which means that some degree of previously agreed mutual knowledge is indispensable. The more they share with the other, the more integrated they are and the easier interoperability becomes, but the greater coupling gets.
What the interoperability framework expressed by Table 1 provides is a classification that allows us to better understand the coupling details, namely at what levels it occurs and what is involved in each level, instead of having just a blurred dependency notion. In this respect, it constitutes a tool to analyze and to compare different integration models and technologies.

REDUCING RESOURCE COUPLING WITH COMPLIANCE AND CONFORMANCE

Coupling is thus a necessary evil, without which no interaction is possible. Our goal is to minimize it as much as possible, down to the minimum level that ensures the level of interaction required by the resources to integrate. We will use Figure 6 to understand what can be done with respect to current integration technologies.

Figure 6. Interaction styles: (a) document; (b) document-based service; (c) compliance-based service

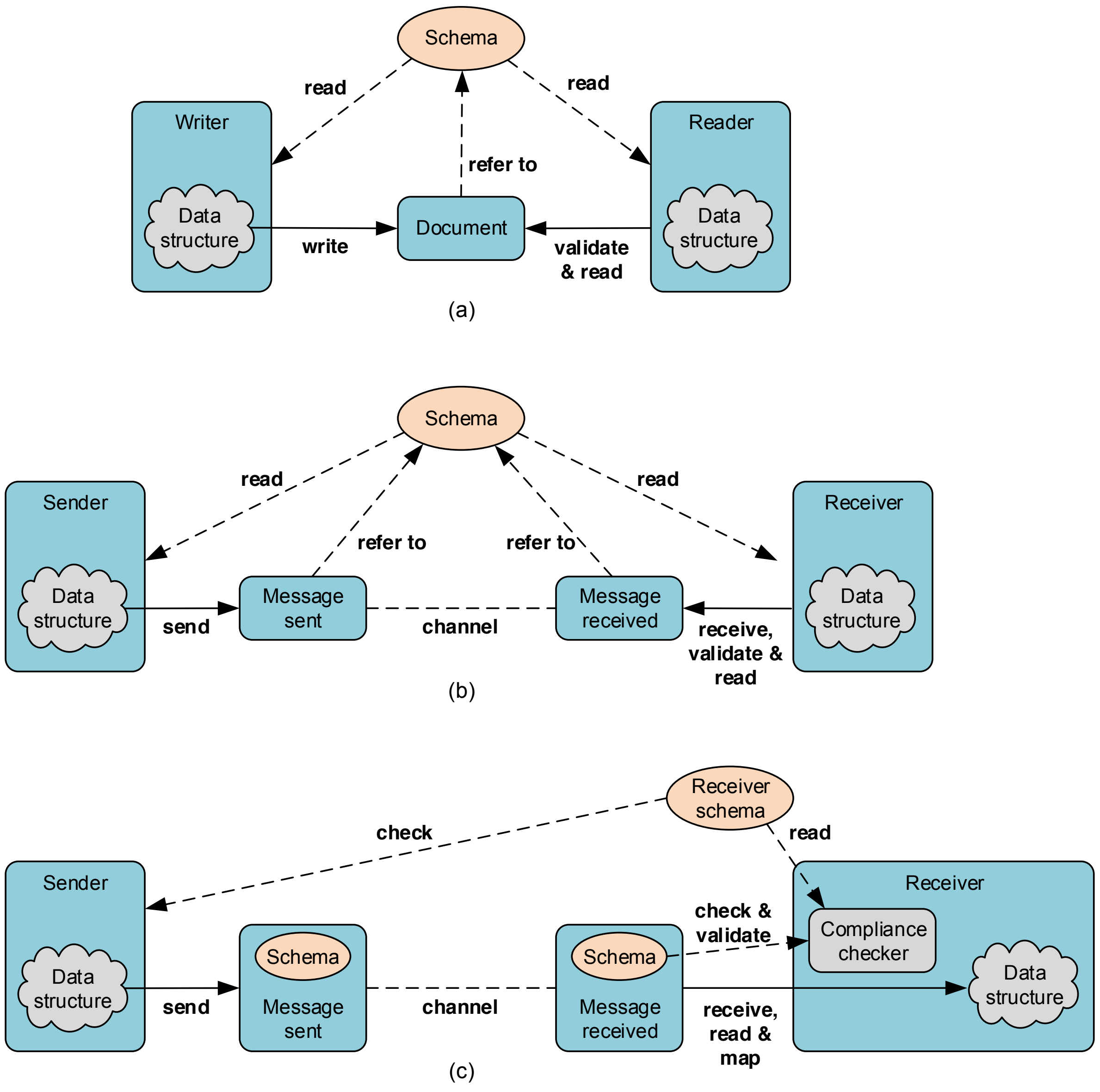


Figure 6a illustrates the classical document-style interaction. This is a symmetric arrangement in which a writer produces a document according to some schema and the reader uses the same schema to validate and to read the contents of the document. There is no notion of service, only the passive resource that the document constitutes. We are at the level of data description languages, such as XML or JSON.
Figure 6b introduces services, in which a message is sent over a channel and received by the receiver. It is now treated as a parameter to be fed to some behavior that the receiver implements, instead of just data to be read. However, the message is still a document, validated and read essentially in the same way as in

Figure 6a. We are at the level of Web Services or RESTful resources. The behavior invoked can thus be exposed and implemented in various ways, but in the end the goal is similar. It is important to recognize that the schemas in Figure 6 refer to type specifications and need not be separate documents, as it is usual in XML Schema and WSDL. In the REST world, schemas are known as media types but perform the same role. The difference is that, instead of being declared in a separate document referenced by messages, they need to be previously known to the interaction resources, either by being standard or by having been previously agreed. In any case, the schema or media type must be the same at both sender and receiver and therefore imposes coupling between the resources for all its possible values, even if only a few are actually used.

Figure 6c depicts a different approach, based on *compliance*. Messages do not obey some external schema. Each message has one specific value (most likely structured, but it is not a type, only one specific value), with its own exclusive schema that is nothing more than a self-description, without the value variability that a type exhibits. This value and its description can be validated against an infinite number of schemas, those that have this particular value included in the set of their instances.

The receiver in Figure 6c exposes a schema that defines the values it is willing to accept. When a message is received, its internal schema is checked against the receiver's own schema. If it *complies* (satisfies all the requirements of the receiver's schema), the message can be accepted and processed. The advantage of this is that a resource can send a message to all the resources with schemas that the message complies with and, conversely, a resource can receive messages from any resource that sends messages compliant with its receiving schema.

In other words, coupling occurs only in the characteristics actually used by messages and not in all the characteristics of the schemas used to generate the message or to describe the service of the receiving resource. Since the schemas of the message and of the receiver are not agreed beforehand, they need to be checked *structurally*. Resources of primitive types have predefined *compliance* rules. Structured resources are compared by the names of components (regardless of order of declaration or appearance) and (recursively) by the compliance between structured resources with matching names. Since the order of appearance of named component resources may different in the message and in the receiving schema, there is the need to map one onto the other (bottom of Figure 6c).

This is a form of polymorphism that increases the range of applicability of both sender and receiver, constituting a means to reduce coupling to only what is actually used. Sender and receiver no longer need to be designed for each other but, as long as compliance is ensured, one resource can replace another (as a result of a new iteration of the lifecycle, for example). In this case, what is involved is *conformance* between the replacement and the resource replaced, stating that the former supports all the characteristics supported by the latter. When a resource is able to interact with another, although not entirely interoperable with it, we say that have *partial interoperability*.

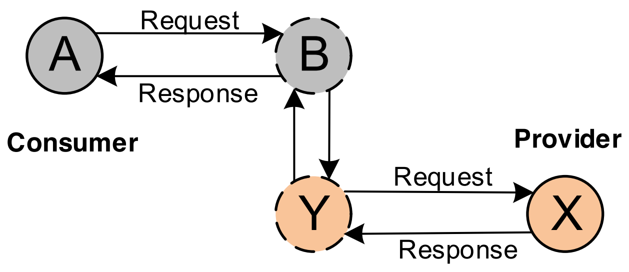
Figure 7 illustrates these concepts and differentiates compliance from conformance. It also describes complete transactions, with both request and response. Figure 6 only describes one message, but it applies both to the request and to the response part of the transaction. The interacting resources now perform the roles of *consumer* (the one making the request) and *provider*, with sender and receiver roles reversed from request to response.

Interoperability of a consumer with a provider is possible by satisfying 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 the characteristics that the provider exposes. This is what happens when schemas are shared. In partial interoperability, the consumer uses only a subset of those characteristics, which means that compliance and conformance need only be ensured for that subset. These properties are not commutative (e.g., if *P* complies with *Q*, *Q* does not necessarily comply with *P*), since the roles of consumer and provider are different and asymmetric by nature, but are transitive (e.g., if *P* complies with *Q* and *Q* complies with *R*, then *P* complies with *R*).

Figure 7. *Partial interoperability, based on compliance and conformance*.



In Figure 7, 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 characteristics that *B* offers and *B* offers only the characteristics 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*. Ensuring that *A* is interoperable with *X* requires two conditions:

- **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 characteristics that *A* uses as a subset of the set of characteristics offered by *X*. This inclusion relationship, without changing characteristics, is similar in nature to the inheritance-based polymorphism supported by many programming languages, but here it applies 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.

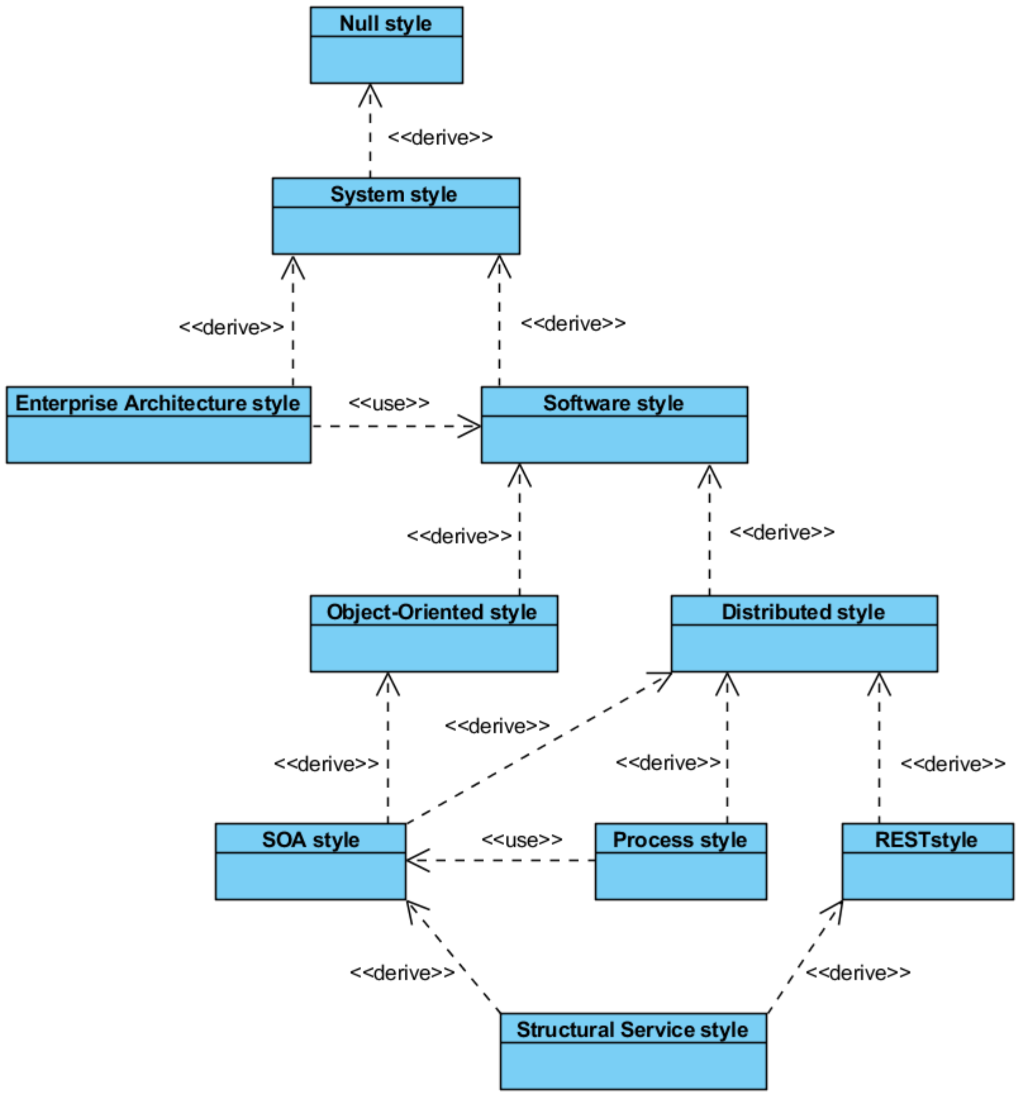
ARCHITECTURAL STYLES FOR INTEGRATION

Armored with frameworks that help in designing resources and in endowing them with the ability to meaningfully interact with others with minimizing coupling, we can now tackle the issue of which architectural style to adopt when trying to achieve resource integration. The main architectural styles (Dillon, Wu & Chang, 2007) used today in distributed systems are SOA (Earl, 2008) and REST (Webber, Parastatidis & Robinson, 2010).

A hierarchy of architectural styles

An architectural style can be seen as a collection of design patterns, guidelines and best practices (Dillon, Wu & Chang, 2007), in a bottom-up approach, or as a set of constraints on the concepts and on their relationships (Fielding & Taylor, 2002), in a top-down approach. We follow the latter because we believe that a rationale is needed before recognizing good 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 8.

Figure 8. A hierarchy of architectural styles.



In this diagram, each class represents an architectural style, which has a derivation dependency on the architectural style above by imposing some constraints. An architectural style can also use elements of another, as illustrated by the two use dependencies.
At the top, we have what Fielding & Taylor (2002) designated the Null style, which corresponds to asserting that there are no constraints at all, not even the existence of the entities depicted in Figure 3. The other architectural styles progressively refine and specialize the Null style by including additional constraints, while reducing the spectrum of entities to which they can apply. For example, the System style identifies the existence of resources of any kind, including software applications, people,

organizations and other non-computer based entities, but the Software style assumes that all resources are software components, developed according to a lifecycle such as the one depicted in Figure 4.
The Software style is further specialized by the Object-Oriented and Distributed styles, which in turn are the basis for the most used architectural styles in the Web and cloud contexts, the SOA (Service-Oriented Architecture) (Earl, 2008) and REST (REpresentational State Transfer) (Webber, Parastatidis & Robinson, 2010) styles. The Process style, with workflows that invoke services, is also rather popular, in particular in business applications (Marjanovic, 2005).
The Enterprise Architecture style illustrates composition of architecture styles and shows that an architecture under consideration does not have to be completely modeled according to a single, pure style. The same happens with the Process style, which is not complete and needs to resort to the Service-Oriented style, in the sense that processes require services as the basic units of behavior.
On the other hand, one style can be a specialization of more than one style, as illustrated by the Service-Oriented style, which combines object oriented and distribution constraints, and the Structural Service style, proposed in this chapter, which specializes both the Service-Oriented and Resource styles. Multiple derivation in Figure 8 applies to constraints, with the less stringent one passed on to the derived style if a constraint is required by both base architectural styles. The goal is to obtain the best of both worlds. For example, with respect to the Structural Service style (further described in the next section):

- The SOA style allows any number of operations for each resource, whereas REST specifies that all resources must have the same operations. In this case, the flexibility of SOA in services is passed on to the Structural Services style;
- The REST style allows structured resources and references (links), whereas SOA does not. In this case, the flexibility of REST regarding resources is passed on to the Structural Services style.

The minimization of coupling provided by compliance and conformance, described in the previous section, is a feature unique to the Structural Services style.
Table 2 summarizes the main characteristics of all these architectural styles. Delgado (2012a) presents a more detailed description of each one.

Table 2. Main characteristics of the various architectural styles.

Style	Brief description	Examples of characteristics or constraints
Null	No style	No constraints, model not applicable, no model elements identifiable
System	Identifies resources, services and processes (model of Figure 3)	Resources are discrete (individualized) Artifacts identified by references Services described in terms of reactions Service interact by stimuli (can be analog and continuous)
Enterprise Architecture	Resources can be people, software and physical	Governance and strategy most relevant Socio-economic factors Processes adequate for human workflow
Software	All resources are software related	Resources are digital Stimuli are finite-sized and time-limited messages Services 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 resources 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 customizable 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, customizable services, minimum coupling	Resources are structured (like REST) Services have customizable interface (like SOA) Interoperability based on structural compliance and conformance (new feature)

Comparing SOA and REST

SOA and REST are the most popular today in distributed integration and the ones we want to compare, between them and against the Structural Services style. The first aspect to realize is that comparing architectural styles in abstract terms is not at all the same thing as comparing their practical instantiations. SOA is easy to grasp, since it constitutes a natural evolution of the object-oriented style, which analysts and programmers are already familiar with. However, Web Services, with WSDL, XML Schema and SOAP, are complex to use, especially without adequate tools that usually automate or hide most of the development process. On the other hand, REST is comparatively harder to master, in particular for people with an object-oriented mindset, who tend to slide to the probably more familiar RPC (Remote Procedure Call) style (Fielding, 2008), with operation parameters instead of dynamic resource references. On the other hand, its use with HTTP is relatively straightforward, even without elaborate tools, since what is involved is essentially plain HTTP messaging.
We are more interested in the architectural styles themselves than on their instantiations, since these are heavily dependent on syntactic and protocol issues of existing integration technologies. We want to compare styles, not their instantiations, since a style can be implemented in several ways.
Both SOA and REST have advantages and limitations. Our goal is to derive an architectural style that emphasizes the advantages and reduces the limitations. The best way to achieve this is to understand the motivations behind the appearance of each style:

- SOA appeared in the context of enterprise integration, with the main goal of achieving *interoperability* between existing enterprise applications with the emerging XML-based technologies, more universal than those used in previous attempts, such as CORBA and RPC (Gray, 2004). It was a natural evolution of the object-oriented style, now in a distributed environment and with large-grained resources to integrate. Therefore, the emphasis was put in modeling applications as black boxes, by emphasizing their external functionality with an interface composed of a set of operations, application dependent. Internal structure was avoided, as measure to reduce coupling by applying the information hiding principle;
- REST appeared in the context of Web applications, as a systematization of the underlying model of the Web and with *scalability* as the most relevant goal, since interoperability had already been solved by HTML (and, later, XML) and HTTP. As proposed by Fielding (2000), REST is based on six constraints, of which *stateless interactions* (only the client maintains the interaction state) and *uniform interface* (all resources have a service with the same set of operations, with variability on structure and with links to resources, rather than on service interface). These constraints have one single motivation: *decoupling* to support scalability. When potentially

thousands of clients connect to a server, we want server applications to scale and to evolve as needed and as independently as possible from the clients.

It is clear that both SOA and REST tackle the fundamental problem of integration: to achieve interoperability while trying to minimize coupling. In addition, both try to comply with the information hiding principle by hiding the implementation of individual resources. However:

- SOA's approach is to hide resource structure (considering it part of the implementation), whereas REST's approach is to hide resource behavior (its service) by converting it into resource structure (each operation is modeled as a resource);
- SOA tries to minimize the semantic gap (Ehrig, 2007) by modeling resources as close as possible to real world entities. Since these are different from each other, offering a specific set of functionalities, the result is a set of resources with different interfaces, with a different set of operations. The consumer must know the specific interface of the provider. This means coupling and may only be acceptable if the number of consumers for a given provider is usually reasonably small and the system evolves slowly (to limit the maintenance effort);
- The interface coupling is precisely what REST tries to avoid, with a simple idea: to decompose complex resources into smaller ones (with links to other resources), until they are so primitive and/or atomic that they can all be treated in the same way, with a common interface. The initial complexity is transferred to the richness of the structure of the links between resources. The most distinguishing feature of REST is this uniform interface for all resources, with a common set of operations. This corresponds to separating the mechanism of traversing a graph (the links between resources) from the treatment of each node (resource). The overall idea is that a universal link follow-up mechanism coupled with a universal resource interface leads to decoupling between server and clients, allowing servers to change what they send to the clients because they will adapt automatically, by just navigating the structure and following the links to progress in the interaction process;
- SOA-based processes are modeled as workflows of activities at the server, invoking specific operations on services along the way. REST-based processes are modeled as state machines at the client, transitioning from one resource representation to another. In fact, the execution models of SOA and REST can be shown to be dual of each other (Delgado, 2012a);
- SOA has no dynamic visibility control mechanism for operations. In contrast, REST limits what the client can do to the links contained in the resource representations retrieved from the server, which helps to increase the robustness of applications. The idea of providing only the currently allowable commands has been borrowed from the field of user interfaces and is known as *affordances* (Amundsen, 2012). This is a strong point in favor of REST.

Unfortunately, both SOA and REST have fallacies and limitations:

- SOA assumes that all resources are at the same level and that exposing their services is just providing some interface and the respective endpoint. The fallacy is to assert that resource structure is unimportant. This may be true when integrating very large-grained resources, but not in many applications that include lists of resources that need to expose their services individually. The limitation is in changeability. Changing the interface of the provider is most likely to require changes in the consumers as well;
- REST imposes a uniform interface, which allows treating all the resources in the same way. The fallacy is to consider only a syntactic interface, or even semantic, with SA-REST (Sheth, Gomadam & Lathem, 2007), and forgetting about the pragmatic and symbiotic levels (Table 1). The reaction of resources need to be considered, therefore following a link cannot be done blindly. The client needs to know which kind of reaction the server is going to have. In the same

way, traversing a structured resource implies knowing the type of that resource, otherwise we may not know which link should be followed next. This only means one thing: since there is no declaration of resource types, all types of resources to be used must be known in advance, either standardized or previously agreed (with custom media types). This is a relevant limitation of REST. What happens in practice is that developers are more than happy to agree on simple resource structures and an aligned API to match. REST advocates contend that an API should be represented simply by its initial link (an URI, for example) and all the functionality should be dynamically discovered by the client by exploring links. This can only be done with a set of fixed resource types (predefined or previously agreed). Resource structure can vary, a good advantage of REST, but only within the boundaries of previously agreed resource types.

In summary, SOA is a good option for large-grained, slowly-evolving complex resources. REST is a good option for small-grained, structured resources which are relatively simple. In search for simplicity and maintainability, many providers (including cloud computing providers, such as Rackspace) have stopped using SOA APIs in favor of REST, by modeling operations as resources, using naturally occurring lists of resources as structured resources with links and dynamically built URIs instead of operation parameters. Decoupling in REST, however, is not as good as its structure dynamicity seems to indicate. All the resource types (media types, in REST and Internet parlance) must be known in advance and the client may break if a new media type unknown to the client is used by the server. In addition, even when the type is known, the client cannot invent what to do and which link to choose when it receives a resource representation of a given type. The pragmatic layer (Table 1) requires behavior and its effects to be implemented at both the server and the client, and these cannot be agnostic of each other.

THE STRUCTURAL SERVICES ARCHITECTURAL STYLE

Main characteristics

In a local environment (application), the object-oriented paradigm is probably the best in the non-declarative world, with its low semantic gap, information reuse (inheritance), polymorphism and ability of exposing both behavior and structure. Unfortunately, in a distributed environment pointers to objects can no longer be used and inheritance stops working, since the lifecycles of the objects are not synchronized. Remote Procedure Call (Gray, 2004) was an early attempt to transport the object-oriented paradigm to the distributed world, but failed due to a very poor interoperability model, language specific. SOA was a much better interoperability solution (thanks to XML), but forgot that classes could expose structure and had polymorphism. REST went in the opposite direction, exposing only structure (actually, converting operations into resources, or behavior into structure), but also without a solution for polymorphism (media types need to be previously known).
What we would like to have is a solution that allow resources:

- To refer to other resources by distributed references (e.g., URIs);
- To return a description of itself, under request;
- To expose operations, as needed for a low semantic gap in complex resource modeling;
- To expose structure, to easily model resources that are naturally structured;
- To use structural (rather than inheritance-based) polymorphism, for increased interoperability, adaptability and changeability, without the need to have resource types necessarily shared or previously agreed.

In a sense, we would like to have the object-oriented paradigm back, but now with distributed references, structural polymorphism and self-description. This is the essence of the *Structural Services* architectural style proposed by this chapter.

The proposal of a new architectural style assumes that we know how to answer the question of what do we gain with it or, equivalently, which limitations of current architectural styles (namely, SOA and REST) does it solve. The previous discussion, including the critique on SOA and REST, already sheds light into this matter. Table 3 summarizes it.

Table 3. Comparison between SOA, REST and Structural Services.

	SOA	REST	Structural Services
Basic tenet	Behavior	Hypermedia (structure + links)	Tunable between pure behavior and pure hypermedia
Distinguishing features	Resource-specific interface; Operations are entry points to a service; Design-time service declaration	Uniform interface; Operations are resources; Clients react to structure received (do not invoke resource interfaces)	Variable resource interface; Resources are structured; Operations are resources; Links are resources; Resource types need not be known at design time
Best applicability	Large-grained resources (application integration)	Small-grained resources (CRUD-oriented APIs)	Wide range (small to large, behavior to structured-oriented)
Interoperability based on	Schema sharing	Predefined media types	Structural polymorphism (compliance and conformance)
Self-description	Repository (e.g., WSDL document)	Content type declaration in resource representations	Included in each resource, message or application resource
Main advantages	Low semantic gap (resources model closely real world entities)	Structured resources, with links to other resources (hypermedia)	Low semantic gap + structured resources with links + polymorphism + self-description
Main fallacy	Resource structure is unimportant	Hypermedia increases decoupling	None identified
Main limitation	No polymorphism (coupling higher than needed)	Fixed interface (semantic gap higher than needed)	None identified

A simple implementation example

For now, the Structural Services architectural style is not much more than a proposal, since it has not been tried out yet in significant scale to confirm its claims. SOA and REST have many working instantiations, which is not the case of Structural Services. However, these are limitations inherent to the development of a new architectural style, not to its intrinsic capabilities.
How can we provide an instantiation of the Structural Services style? There are some possibilities, including:

1. To extend WSDL files with another section (which we could call Structure), in which component resources and links could be exposed, thereby allowing Web Services to have structure;
2. To allow REST resources to have their own service description (e.g., WSDL file), thereby allowing the coexistence of the uniform and non-uniform interface approaches in REST;
3. To define a new resource description language, conceived specifically for the Structural Services architectural style and providing native support for its characteristics.

Solutions 1 and 2 are certainly possible, but existing tools are not designed to support the new style, which means that the benefit of using readily available software would really not exist or at least hampered by practical limitations. Data compliance and conformance are also possible with data description languages such as XML and JSON (these are data concepts that do not depend on notation or format), but again the available middleware does not support this kind of data mapping. In addition,

compliance and conformance in Structural Services entail not only data but also operations, which neither XML nor JSON support.
Therefore, we had to define and implement a language (SIL – Service Implementation Language) to describe resources and their services (Delgado, 2012b; Delgado, 2013), in accordance with the new style. Space limitations allow us just to present a very simple example, to provide a look and feel of its possibilities. We describe a car rental application, with the server side in Listing 1 and the client side in Listing 2.

Listing 1. Provider (server) side of the car rental application.

```
myCarRental: {
   date: definition {day: [1..31]; month: [1..12]};
   carClass: definition union {"economy"; "compact"; "standard"; minivan"};
   carInfo: definition {class: carClass; rate: float; linkCar: link to car};
   car: definition {
      class: carClass;
      occupied: list of date;    // days already reserved
      rate: float;               // daily rate
      operation (-> carInfo) { reply {class; rate; self}; };
      reserve: operation (day: date ->) { occupied.add <- day };
      cancel: operation (day: date ->) { occupied.remove <- day };
   };

   fleet: list of car;  // exposed resource structure

   getFleet: operation (class: carclass:=any -> info: list of carInfo) {
      for (i: [0..fleet.size-1]) {
         if (fleet[i].class ==> class) // if the car's class complies
            info.add <- (fleet[i]<-);  // invoke the nameless operation in "car"
      reply info;
   };

   getFleetSize: operation (-> int) {
      reply fleet.size;
   };
};
```

Listing 1 describes a resource, `myCarRental`, the online application of a car rental enterprise. A resource in SIL has the look and feel of a JSON data structure (components with braces, `{...}`), but component names are not strings and there are operations with code.
The first declarations (`date`, `carClass`, `carInfo` and `car`) are definitions, not actual resources, and their purpose is to avoid repeating the definitions whenever a resource with these structures need to be created. Each time they are used, a resource with that structure is created. The keyword `union` means that `carClass` can have one of the string values indicated.
The first real resource component, `fleet`, is a list of resources describing the various cars that compose the fleet. Note that these, described the `car` definition, have three data resource components and three operations, one without name and two with names. The component `fleet` is accessible from outside `myCarRental`, by using the resource path `myCarRental.fleet`.
Operations are first-class resources and can be accessed by a resource path and be sent messages (with the operator "`<-`", just like any other resource. Adding a new car to the fleet can be done by using the exression `myCarRental.fleet.add <- ...`, sending the description of the new car as message argument. This is also illustrated in Listing 2, with access from the client side.
When a structured resource is sent a message, an operation with an argument which the message complies with is searched among the operations of the resource (which is then invoked, if it exists). If the receiving resource is an operation, then only that operation can be invoked and the message must comply with the

operation's input argument (otherwise, an exception occurs). Operations can only have one input argument and one reply value, but these can be arbitrarily structured. In the operation's declaration heading, they are separated by "->". Any of them may be non-existent.
Each car offers the functionality of providing information on it (by sending it a message without an argument, as illustrated in operation getFleet in myCarRental) and this information includes the car's class and rate, as well as a link that can later be used to perform operations on the car description in the fleet, in the hypermedia style heralded by REST. The keyword self indicates a reference to the resource that contains the operation in which it appears.
The operation getFleet returns a list with information (class, rate and a link to each car descriptor in the fleet) on each car, allowing the client to navigate through this list and use the link of the chosen car to make the reservation, as illustrated by Listing 2, in which the first car with daily rate below 40 is reserved. Note the heading of getFleet in Listing 1, which declares an input parameter (class) that is optional and has value any by default (specified by the operator ":="). This means that this argument gets the value passed, is specified, or any, if omitted. Then, the compliance check operator ("==>") indicates whether each car's class complies with the class required. If omitted, and since all resources comply with any, by definition, the entire fleet is returned.

Listing 2. Consumer (client) side of the car rental application.

```
mcrSPID: definition {
   date: definition {day: [1..31]; month: [1..12]};
   carClass: definition union {"economy"; "compact"; "standard"; minivan"};
   carInfo: definition {class: carClass; rate: float; linkCar: link to car};
   car: definition {
      class: carClass;
      occupied: list of date;    // days already reserved
      rate: float;               // daily rate
      operation (-> carInfo);
      reserve: operation (date ->);
      cancel: operation (date ->);
   };
   fleet: list of car;  // exposed resource structure
   getFleet: operation (carclass:=any -> list of carInfo);
   getFleetSize: operation (-> int);
};

mcr: link to mcrSPID;      // initialized with a link to myCarRental

mcr.fleet.add <- {class: "economy"; rate: 30};     // add cars
mcr.fleet.add <- {class: "standard"; rate: 50};

numberOfCars: mcr.getFleetSize <-;    // invokes operation getFleetSize
carFleet: mcr <-;                     // invokes operation getFleet and returns
                                      // the entire fleet (class in getFleet
                                      // is "any" by default)
minivanCars: mcr <- "minivan";        // returns only the fleet of minivans
smallCars: mcr <- union {"economy"; "compact"};    // returns only the fleet of
                                      // economy and compact cars

for (j: [0.. smallCars.size-1]) {
   c: smallCars [j];           // car information. Brings a link to the resource
   if (c.rate < 40){
      c.linkCar <- {23; 10};    // Use the link to reserve the car for October 23
      break;
};
```

Listing 2 illustrates the usage of the myCarRental application, from the client side. We have included the SPID (SIL Public Interface Descriptor), which performs the same role as WSDL files (to provide a description of the interface). Unlike WSDL, SPIDs are generated automatically from the program, by eliminating private parts such as the body of operations. This is akin to SOA style, since the client can obtain this description from some resource/service registry. But this is also equivalent to declaring a media type in REST, which must be previously agreed by both client and server.
We use a link to the server application (`mcr`), which must be obtained by the client somehow (specified in the program or found in a registry). In SIL, a link is not necessarily a URI. It is a resource with structure previously agreed between client and server. A URI is just one of the possibilities.
We illustrate how car descriptors can be added, to grow the fleet, and how to retrieve a list of information structure on each car, as described above, both for the entire fleet and just for some of car classes. The operation invoked is the same and all is based on compliance. Finally, a small loop runs through the list and makes a reservation for October 23, for the first car with a daily rate under 40.
Naturally, these are very simple examples and many things are not tested, such as whether the car is free on this day. In a more complete and robust program, these tests would have to be performed.
SIL is a compiled language. The text source is compiled into binary form, which is used for message transmission and program execution, with an interpreter. Connection to other languages (Java, C#, Python, and so on), is done by the usual mechanisms. There is a mapping between primitive resource types in SIL and each programming language, an adapting layer and a dynamic class load mechanism to support changes to resources.
The existence of the myCarRental's SPID n Listing 2 is not essential. At the client, `mcr` could have been declared as `any`, which switches off compiler verifications and defers compliance checks entirely to runtime, which is done structurally and without previous knowledge of the resource types. This is much slower than SPID-based compile-time verifications but, after the first time of each compliance check, the corresponding resource structural mapping (Figure 6c) can be cached and reused in subsequent messages.
Regarding the status of implementation of SIL, there is a prototype with basic functionality, with a compiler based on ANTLR (Parr, 2007) and an interpreter written in Java, with Web Sockets (Lubbers, Albers & Salim, 2010) as the transport protocol and a Jetty-based server to support resource links and messaging. Delgado (2013) provides a more detailed description of the language and of its implementation.

FUTURE RESEARCH DIRECTIONS

The approach that we have followed, proposing the Structural Services architectural style, which can bridge the characteristics of SOA and REST and increase decoupling and the range of applicability, has the potential to improve substantially the current technological scenario, but much remains to be done before the architectural style is realistically tested and its capabilities demonstrated.
Future work will mainly entail the following activities:

- Completing the implementation. For example, the language SIL supports concurrency and asynchronous communication, based on futures, and delegation, but these features are not implemented yet. Semantics is another aspect to complete. Instead of using annotations or specific semantic languages, SIL supports semantics through structural ontology definitions and rule resources (set of condition-then-action statements), which become active upon creation in the scope of their container resource and are reevaluated 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 is designed but not implemented yet;
- The current compliance and conformance algorithms are implemented just at design time, by the compiler. These algorithms also need to be implemented at the binary level to be checked at

message arrival time, with cache-based mechanisms to optimize message exchange and dynamic resource type checking. The effectiveness of these mechanisms needs to be assessed;

- 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 in each of the cells in Figure 5 and in each layer of Table 1;
- The interoperability framework presented in this chapter needs to be improved and made more complete, namely in the Concerns axis, to include additional concerns such as security and common domain-specific aspects and problems. Many have already been uncovered by other frameworks, namely by the project ENSEMBLE (Jardim-Goncalves, Agostinho & Steiger-Garcao, 2012), which is systematizing the interoperability domain, establishing a scientific base for enterprise interoperability and developing an interoperability body of knowledge (Jardim-Goncalves et al, 2013);
- Carrying out a comparative study, with qualitative and quantitative assessment, between the Structural Services architectural style, with is instantiated with the SIL language and platform, and SOA and REST, with their instantiations with Web Services and HTTP-based interfaces, respectively.

CONCLUSIONS

Enterprise integration is an essential problem to deal with, since enterprises need to interoperate to be able to pursue common or complementary goals. We have defined the fundamental problem of resource integration as the goal of providing the maximum possible resource decoupling while ensuring the minimum interoperability requirements.

SOA and REST are the most common architectural styles that try to address this problem, but focusing more on the interoperability than on the decoupling side of the issue. Both require previous knowledge of the types of the resources involved in the interaction. SOA emphasizes behavior and does not contemplate resource structure. REST emphasizes structure and is not a good modeling match for complex resources, since these must be decomposed into simpler resources with a uniform syntactic interface (but different at the semantic, pragmatic and symbiotic levels).

Our approach is twofold:

- On the decoupling side of the integration problem, our solution is to use structural compliance and conformance as the basis for relaxing the constraint of having to share resource types, previously known. Any two resources that share *at least* the characteristics required for interoperability can interoperate, independently of knowing the rest of the characteristics, at design, compile or runtime. This is maximizing decoupling, something that neither SOA nor REST achieve;
- On the interoperability side, syntax or even semantics are not enough. Higher levels (pragmatic and symbiotic) need to be considered as well. This means that the principle of uniform interface of REST (trying to deal with resources in an universal way) is an illusion, achievable only at the syntactic level. The result is that REST is also an incomplete solution for integration. But resource structure is fundamental. In real life, most entities are naturally structured and applications depend on exposing and accessing that structure. Therefore, any integration solution must natively support resource structure. This means that SOA is an incomplete solution to integration, too. To have a complete solution, we need to combine characteristics of SOA and REST and complement with others.

We have described and proposed a new architectural style for integration, Structural Services, which combines the behavior flexibility of SOA with the structural hypermedia capabilities of REST and structural interoperability based on compliance and conformance.
We contend that these are features are essential to the solution of the fundamental integration problem, resulting in a low modeling semantic gap (resources closely model real world entities, both in structure and behavior), while maximizing decoupling for high adaptability, changeability and reliability.
We have also briefly discussed SIL, a service-oriented distributed programming language that provides an instantiation of the Structural Services architectural style.
We hope that, in due time and with further developments, the proposals made in this chapter can contribute to developing a simpler alternative to Web Services and a higher modeling level alternative to RESTful interfaces and applications, with a wider range of applicability.

REFERENCES

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.

Ahmad, M., & Bowman, I. (2011). Predicting System Performance for Multi-tenant Database Workloads. In Fourth International Workshop on Testing Database Systems (pp. 6.1-6.6). ACM Press.

Amundsen, M. (2012, April). From APIs to affordances: a new paradigm for web services. In *Proceedings of the Third International Workshop on RESTful Design* (pp. 53-60). ACM Press.

Chandler, D. (2007). *Semiotics: the basics*. New York, NY: Routledge.

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

Delgado, J. (2012a). Bridging the SOA and REST architectural styles. In Ionita A., Litoiu, M. & Lewis, G. (Eds.) *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments* (pp.276-302). Hershey, PA: IGI Global.

Delgado, J. C. (2012b). Distributed Service Programming and Interoperability. In Loo, A. (Ed.) *Distributed Computing Innovations for Business, Engineering, and Science* (pp. 111-136). Hershey, PA: IGI Global.

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.

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.

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

Ehrig, M. (2007). Ontology alignment: bridging the semantic gap (Vol. 4). New York, NY: Springer Science+Business Media, LLC.

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 Qualitive and Quantitative Methods in Libraries* (pp. 345-352). Singapore: World Scientific Publishing Company.

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

Fielding, R. (2008). REST APIs must be hypertext-driven. *Roy Fielding's blog: Untangled*. Retrieved November 11, 2013, from http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

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

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

Gray, N. (2004, April). Comparison of Web Services, Java-RMI, and CORBA service implementations. In Schneider, J. & Han, J. (Eds.) *The Fifth Australasian Workshop on Software and System Architectures* (pp. 52-63). Melbourne, Australia: Swinburne University of Technology.

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

Greefhorst, D., & Proper, E. (2011). *Architecture principles: the cornerstones of enterprise architecture*. Berlin Heidelberg: Springer-Verlag

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

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

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.

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.

Kruchten, P. (2004). *The rational unified process: an introduction*. Boston, MA: Pearson Education Inc.

Li, L., & Chou, W. (2010). Design Patterns for RESTful Communication. In *International Conference on Web Services* (pp. 512-519). Piscataway, NJ: IEEE Computer Society Press.

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 Computer Society Press.

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

Malik, N. (2009). Toward an Enterprise Business Motivation Model. *The Architecture Journal,* 19, 10-16.

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

Minoli, D. (2008). *Enterprise Architecture A to Z*. Boca Raton, FL: Auerbach Publications

Oguz, F., & Sengün, A. (2011). Mystery of the unknown: revisiting tacit knowledge in the organizational literature. *Journal of Knowledge Management*, 15(3), 445–461.

O'Rourke, C., Fishman, N. & Selkow, W. (2003) *Enterprise architecture using the Zachman framework*. Boston, MA: Course Technology.

Palm, J., Anderson, K., & Lieberherr, K. (2003). *Investigating the relationship between violations of the law of demeter and software maintainability*. Paper presented at the Workshop on Software-Engineering Properties of Languages for Aspect Technologies. Retrieved November 11, 2013, from http://www.daimi.au.dk/~eernst/splat03/papers/Jeffrey_Palm.pdf

Parr, T. (2007). *The Definitive ANTLR Reference*. Raleigh, NC: The Pragmatic Bookshelf.

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. "big"' web services: making the right architectural decision. In *International conference on World Wide Web* (pp. 805-814). 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 *International Conference on Service-Oriented Computing and Applications* (pp. 1-4). Piscataway, NJ: IEEE Computer Society Press.

Shadbolt, N., Hall, W., & Berners-Lee, T. (2006). The semantic web revisited. *IEEE Intelligent Systems*, 21(3), 96-101.

Sheth, A., Gomadam, K., & Lathem, J. (2007). SA-REST: semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6), 91-94.

Uram, M. & Stephenson, B. (2005). Services are the Language and Building Blocks of an Agile Enterprise. In Pal, N. & Pantaleo, D. (Eds.) *The Agile Enterprise* (pp.49-86). New York, NY: Springer.

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 Computer Society Press.

ADDITIONAL READING SECTION

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.

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.

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.

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.

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

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

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

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.

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.

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.

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.

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.

Läufer, K., Baumgartner, G., & Russo, V. (2000). Safe Structural Conformance for Java. *Computer Journal*, 43(6), 469-481. Oxford, UK: Oxford University 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., Peristeras, V., & Tarabanis, K. (2011). Towards a reference service model for the Web of Services, *Data & Knowledge Engineering*, 70, 753–774.

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.

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

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.

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.

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

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.

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: 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: 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: 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*.

Integration: The act of instantiating a given method to design or to adapt two or more resources, so that they become interoperable as a requisite to be able to cooperate and to accomplish one or more common goals.