# MODELING DATA INTEROPERABILITY FOR E-SERVICES

José C. Delgado
*Instituto Superior Tecnico, Technical University of Lisbon, Av. Cavaco Silva, Porto Salvo, Portugal*

Keywords: Service interoperability, Service modeling, Structural conformance.

Abstract: In global, distributed systems, services evolve independently and there is no dichotomy between compile and run-time. This has severe consequences. Static data typing cannot be assumed. Data typing by name and reference semantics become meaningless. Garbage collection cannot be used in this context and (references to) services can fail temporarily at one time or another. Classes, inheritance and instantiation also don't work, because there is no coordinated global compile-time. This paper proposes a service interoperability model based on structural conformance to solve these problems. The basic modeling entity is the resource, which can be described by structure and by behavior (service). We contend that this model encompasses and unifies layers found separate in alternative models, in particular Web Services and RESTful services.

## 1 INTRODUCTION

Complex and distributed information systems are typically built by code in a general purpose programming language, such as Java or C#, wrapped into Web Services and orchestrated by a process based workflow language such as BPEL.

Distributed data interoperability has been a recurring issue, with XML and Web Services based solutions as the main answers to this problem. Higher levels of interoperability have been identified (Tolk, 2006) in the context of the semantic web.

However, XML is essentially a text-based serialization format, conceived mostly as an extensible, customizable and self-describable way of specifying documents, both in content and structure (*schema*), and not a general purpose structured data format designed for e-service interoperability. Binary data is still treated in a separate, special way.

The relevance and widespread use attained in just a few years by web browsing, intrinsically user driven, was enough to justify the option of designing XML as an evolution of HTML (maintaining text-based markup) in what browsing and hypermedia document description is concerned.

E-services and their interoperability, in the same line of thought, became aligned with XML and built on top of it, by using SOAP and the WS-* stack. Unfortunately, this has introduced a level of complexity and overheads that have motivated the appearance of alternatives and variants, such as REST (Pautasso, 2009) and WOA (Thies and Vossen, 2009), in an attempt to build simpler and more efficient systems.

This paper considers the original problem, service interoperability, and draws a model of the requisites it should comply with. This model is then compared with the canonic XML-based solutions available on the market today and the advantages of this model analyzed.

## 2 THE PROBLEM

### 2.1 Data Interoperability

Although we commonly use the term *semantic web*, ontologies are increasingly used and defined, researchers identify several levels of interoperability (Tolk, 2006) and *knowledge transfer* is a popular term (in April 2010, Google Scholar was able to retrieve more than 80,000 entries), the fact is that in the end interoperability boils down to data. All the other levels must build on top of it.

If a service wants to share information or knowledge with another, it can't. What it must do is to produce data from its context, pack it into a message and send it to the other service, which must reinterpret it in its own context. If the ontology is not the same, the information retrieved and the resulting knowledge (or belief) will be different from the original. But common ontologies can only be

C. Delgado J. (2010).
MODELING DATA INTEROPERABILITY FOR E-SERVICES.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Information Systems Analysis and Specification*, pages 271-276
DOI: 10.5220/0002907902710276

established cooperatively by data communication, so no entity can be certain that the message it sends gets correctly understood.

This can only be done by prior agreement between the programmers of both services, using some other form of communication. In fact, the set of primitive data types used in a message format can be considered the lowest common agreed ontology. Both parties need to know what a byte sequence is to exchange at low level a group of bytes, even without further meaning. This paper does not tackle higher levels of service interoperability (e.g., semantics).

## 2.2 Problems in Distributed Services

When programming one application, programmers have the luxury of compiling the entire source program at the same time (separate compilation is just an optimization that avoids compiling modules that do not depend on some change). In other words, the life cycles of all the services contained in that application are synchronized. Figure 1 illustrates a typical life cycle of a service, with two loops. If evaluation (according to some KPIs) is not satisfactory, the cycle is restarted. If there is a change, a new version is built (and the current finalized). If the strategy finds that the service is not worthwhile changing, it is eliminated.

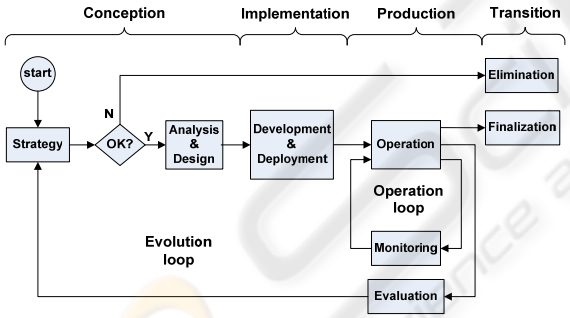


Figure 1: Typical lifecycle of a service.

In global, distributed systems, services evolve independently and there is no dichotomy between compile and run-time. The system is always in run-time and the interface of a service (and even its location, through migration) can change at any time without warning.

This has severe consequences. Static data typing cannot be assumed. In fact, data typing by name and reference semantics, usually basic programming features, become meaningless. Reliable garbage collection becomes much more difficult to achieve in this context and (references to) services can fail temporarily at one time or another. Classes, inheritance and instantiation also don't work, because there is no globally coordinated compile-time. How can we cope with such an environment?

# 3 AN INTEROPERABILITY MODEL

This section presents a simple model, adequate for distributed systems and reflecting our idea of what service data interoperability should be based on.

## 3.1 Structure

We use structured models to capture the essential aspects of structured entities of the real world, in which change is the only constant. Therefore, services must be prepared and agile to cope with changes. The best way of achieving this is to have a model that resembles reality as close as possible, so that a small change in reality translates into a small change in the model. An entity can be modeled by:

- Behavior only (black box approach, described exclusively by how it reacts to stimuli from the outside world);
- Structure only (its behavior depends entirely on the interaction behavior resulting from its internal component entities and how they are interconnected);
- Both structure and behavior.

The main structuring paradigm is composition (in the UML sense). One entity can only be part of another, in a tree-based structuring hierarchy. An entity can reference another outside its context, but with the reference as one of its component entities.

Behavior as a reaction to stimuli implies interaction. In the context of e-services, our model assumes discrete stimuli in the form of electronic messages, which are entities in their own right and under the same model. To interact, two entities must be directly connected at some interaction point or have a third one (that acts as a channel) connecting directly to both. The channel can support addressability, when it connects to many entities, but this is a mechanism built on top of the basic interaction model.

We use the term *resource* to broadly designate the entities that are part of this model and *service* to designate the behavior exhibited by a resource at an interaction point. From the outside, a service appears to be a non-structured resource with one interaction point. A structured resource is a collection of

resources that, when expanded, yields a tree of structured resources with services at the leaves.

Figure 2 illustrates the structure in this model. Each resource can be described as a black box in terms of behavior (the set of messages it is able to respond to at the interaction points and corresponding reactions) or structure (its component resources, including internal connections that allow private component resources to interact without making them visible externally).

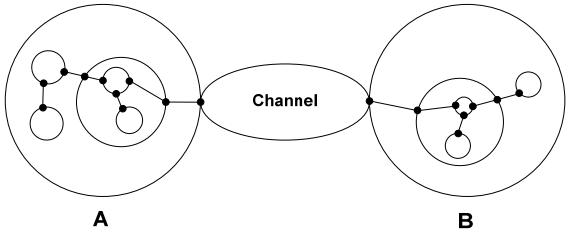


Figure 2: Example of structural modeling. Black dots are the interaction points.

Figure 3 depicts the UML conceptual resource metamodel (there are actually no classes involved). A simple resource has no structure (just the service) and is described only by its behavior, specified as a set of operations.

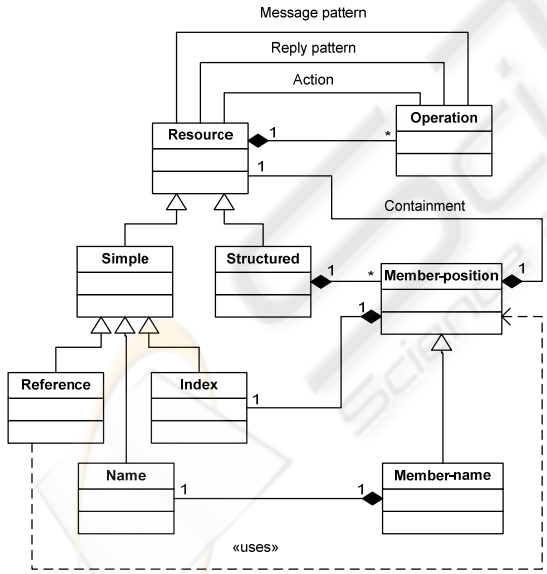


Figure 3: Conceptual resource metamodel.

A structured resource is a collection of member (component) resources, accessible from the containing resource given its index (position) in the collection. Some can also be associated with a name and accessed by it. Members are the interaction points in Figure 2. A visibility (public/private) mechanism can be added to specify which members are accessible from outside a resource.

The index and name would normally be modeled as attributes of the Member classes, but in this way it becomes clearer that we somehow need to model the notions of a non-negative integer and of a string, most likely as primitive resources. The reference is another potential primitive resource, possibly implemented as a URI with a resolution mechanism that supports migration. Having a reference to a resource implies registering it in some resource that offers a directory service.

Therefore, we have two mechanisms to relate resources:

- Containment. Migrating a resource implies migrating all the resources it contains. Its container needs to externalize it, removing it (and its members) from its containing list;
- Usage. An access mechanism (by position and by name) needs to be provided so that, given a resource, a reference to any of its members can be obtained.

## 3.2 Message based Interaction

Messages are themselves resources. If, in Figure 2, a service in resource *A* wants to send a message to a service in resource *B* to invoke some functionality, it needs to:

- Create the message in the context of *A*;
- Externalize it to the channel;
- Internalize it in the context of *B*.

A message is in fact migrated from one resource to another, in whose context it must be understood without requiring additional service specific data that leads to tight coupling.

A *service transaction* is defined as the entire set of operations needed at *A* to produce and send a request, execute it in *B* and eventually send a reply and cope with it back in *A*.

Although higher level interaction patterns are possible (Zdun, Hentrich and van der Aalst, 2006), the basic model of message based service transaction should be asymmetric (who defines the transaction details is essentially the provider and the consumer must comply with this), robust (the consumer must deal with potential faults), push-based (the provider must receive the message on the terms it specifies and not be obliged to scavenge the message in search of a potential request), self-describing, loosely-coupled and asynchronous (e.g., using the future mechanism of some concurrent languages).

## 3.3 Mechanisms for Distribution

Loosely coupled interoperability is one of the main guiding tenets in distributed systems. Both consumer and provider must assume the minimum possible about each other, which precludes the use of many of the features that modern programming languages (in particular, object-oriented) have introduced.
The main solutions provided by our distributed service model are:

- Any resource can change itself dynamically;
- Use of prototypes and cloning instead of classes and instantiation;
- Use of delegation and composition for behavior sharing, instead of inheritance and interface implementation;
- Use of one single, recursive structuring mechanism (the resource);
- Separation of data from the engines that process them, so that services can exchange data but use their own engines;
- Structural conformance (Kim, D., and Shen, W., 2007), to match messages with patterns, based on a structure built out of primitive resources, instead of named typing;
- Use by all interacting services of a low level common abstraction (such as a byte sequence) to be used as a basis for the message format.

## 3.4 The Model in Action

The actual implementation of the resources and their services is their internal affair, depending on the engines they use to implement their behavior.

Figure 4 illustrates the basic operational model of a resource, contemplating members described by behavior (the upper broken line rectangle; only one behavior interaction point represented) and by structure (the lower part; two internal members, with private interaction points, and one with a public interaction point represented). Structurally, this model is recursive, which means that the members in the lower part have a similar structure. The Structure Manager is an engine that supports the topology of interconnections and the access (by name or position) to the resource's members. The set of primitive resources is not really part of the resource and is represented to make apparent that they are available to be replicated (cloned) or shared by using a reference.

The Receive engine includes the message listener and eventually a message buffer. It implements a given transport protocol and deals with the lowest level data abstraction (sequence of bytes).

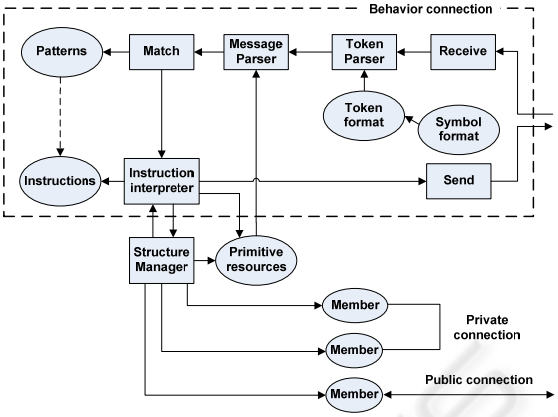


Figure 4: The operational model of a structured resource.

The Token Parser is an engine that reconstructs a sequence of tokens, each representing a non-structured, primitive resource. A symbol format may be used if the underlying alphabet is higher level than bytes (such as characters). The Message Parser engine reconstructs the message as a structure (tree) resource, using primitive resource tokens and structure tokens as input. The Match engine tries to successively match the message with one of the patterns specified in the operations (Figure 3) specified in the resource, using structural conformance. If it succeeds, passes that information to the next engine. Otherwise, it either resends the message to another resource (delegation) or simply ignores the message. The Instruction Interpreter engine reads the (compound) instruction corresponding to the pattern matched and executes actions according to whatever meaning it assigns to what it reads. Possible instructions include replying to the message received or sending a message to another service, which is the job of the Send engine, complementary to the Receive engine.

All these engines need to support concurrency. Reception of a new message creates a new task or thread to carry out the corresponding request. The Receive engine may limit this to avoid saturation of the resource's response capacity. The primitive resources are shared by the various tasks.

## 3.5 Structural Conformance

Sending a message to a (remote) service is, in its essence, a (remote) assignment. In a distributed environment, reference semantics are meaningless or not practical, which means the default must be copy-semantics (reference-semantics is possible with the Reference primitive resource). The value to assign

(the message) is transferred to the context of the resource that receives it and likely affects its state.

One of the basic problems in an assignment is to know whether the operation is acceptable, i.e., if the message conforms to what the receiver expects and knows how to cope with. Here, we cannot compare data types by name. We only have tree structures whose leaves are primitive resources and that are structurally self-describing. Therefore, we need to compare two trees and perform a structural conformance. Specifically, we need to know if a message just received conforms to one of the patterns specified in the receiver service.

A pattern is nothing more than a normal structured resource definition, in which each member is declared as a replica of some other. Named members can have some initial resource value specified. If this is the case, it indicates that this is an optional member for the message. The basic algorithm of structural conformance can be specified recursively in pseudo-code as described below, where *p* and *m* are abbreviations for *pattern* and *message*, respectively, and *x* `<==` *y* is a simple notation for *y conforms to x*. Member names must be unique within each resource. If more than one has a given semantics, they should be encapsulated in a structured resource with that name (and not in an XML style sequence).

```
conforms = true;
for each named member in p
  if (there is a
      m.member.name == p.member.name)
    if (p.member <== m.member)
        p.member = m.member;
     else {conforms = false; break;}
  else if (p.member is optional)
        p.member = default value;
     else {conforms = false; break;}
```

An incoming message will match (conform to) a pattern if all the named pattern members are conformed to, either by message members with the same names (assigned to the pattern members) or by their default values. If there is no match, the message will be tested for conformance against the next specified pattern. In case of match, the resulting (structured) value of the pattern can then be used by the engine executing the (structured) instruction corresponding to that pattern as if it were the real message. Note that the matching is oriented by the pattern that the receiving service expects and not by the message. This means that all the message members that do not match will be ignored. Structural conformance, allowing default values, members out of order and ignoring extra data, is a fundamental aspect in supporting the loose coupling in service interoperability.

This structural conformance algorithm can easily be extended for non-named (by position) members, but space limitations prevent full discussion here.

## 3.6 Describing and Specifying Services

A service should be described by its semantics. However, this is still an insurmountable obstacle for non-trivial services and description is usually limited to a computer-readable interface, complemented with some human-readable comments. We must keep in mind that a service compiler cannot rely on some available service interface description, because the service might have been changed in the meantime. It is more a hint than a specification.

What the consumer may expect as a reply after sending a request message to some service provider is also important, including not only the normal responses but possible exceptions. All these are resources and can be treated as such when processing the reply. Exception treatment can be carried out by some *try-catch* instruction. Structural conformance applies in all cases.

The description of a service can be derived automatically from the source specification of the service itself and consists mainly of the list of message and reply patterns in each operation (Figure 3). A resource is described by its services and, recursively, by its public structured members.

This is just the basic interoperability model. Higher level mechanisms, such as message access control and security, can be implemented by enclosing the message in another message that acts as an outer envelope and that carries the required information (which varies from service to service).

To describe and interoperate resources, we basically need (Figure 4):

- A programmer level language or notation (with structure, message patterns and instructions), out of which interface descriptions can be automatically extracted;
- A set of primitive resources, simple or structured and both data and instructions;
- A serialization format, to support resource migration (and message communication) and processing by the engines;
- A compiler to translate the source specifications to the serialization format;
- A server type of runtime environment to support the engines;
- A transport protocol (at the byte stream level).

# 4 DISCUSSION AND RATIONALE

No system today implements this interoperability model. The most widely available solutions to the e-service interoperability problem are based on Web Services (Peltz, 2003) and REST (Pautasso, 2009). Both are layered approaches. Instead of one model designed for services from ground up, they are based on XML, a document description language, on top of which messages, service descriptions (e.g., WSDL or WADL), behavior (e.g., BPEL) and even protocols (e.g., SOAP) are built as if they were documents described by schema documents. All these layers introduce complexity and overheads.

There is an inherent mismatch here, because XML was not conceived for message based systems. The underlying XML model is symmetric and pull-based, in the sense that an entity produces an XML document and another reads it using typically the same schema. The reader (message receiver in services) then must browse the message in search of what it needs, instead of having it delivered in the format that it expects. That's the difference between a grammar based schema and a pattern based one.

There is also a lack of dynamicity. Data binding takes care of softening (within limits) changes in the schema in what the receiver is concerned. But that usually requires re-compilation.

Our model contemplates document description precisely in the same way as message passing, with the added bonus that data binding is automatic and dynamic, through the mechanism of structural conformance. The receiver copes with a message through its own pattern schema. Which schema the sender used and even if the message complies with it is irrelevant. The only thing that matters is if the message conforms to the schema of the receiver.

Another fundamental issue is the intrinsic difference between the WS and REST styles. The Web Services are usually medium to high granularity and have functionally rich interface. Essentially, they are based on behavior. In contrast, RESTful services are adequate to lower granularities and emphasize a rich structure with a uniform interface. Neither can really change their mode of operation, unlike our model that has the intrinsic ability of tuning up both behavior and structure, by emphasizing structure with simpler interfaces or the other way around. This is a feature granted by the fact that the model is unique and not layered.

Another manifestation of this paradigm is the unification of data and behavior within the model itself. This means that actually programming the behavior of services does not need another layer, as with BPEL, but continues to use the basic service paradigm with a foundation on structural conformance. Active XML (Abiteboul, Benjelloun and Milo, 2008) also contemplates the possibility of invoking Web Services from an XML document, but the model is still document-centered.

Coupled with this issue, and not of lesser importance, lies the mechanism used to produce a schema from a service/resource specification, by simply removing instructions and private resources. If no security issues arise, the schema is simply the public part of the service/resource itself.

# 5 CONCLUSIONS

We have presented a model of service interoperability, based on many of the ideas fostered by the XML structural extensibility and separation of data and processing engines, but in which the basic unit is not the document but the resource, including both structure and behavior (services).

An implementation is under development, tackling the topics mentioned at the end of section 3.6, but not described here due to lack of space. It deals with service interface only, but the model in itself does not hamper semantic conformance (on top of the structural one), which will be pursued next.

# REFERENCES

Abiteboul, S., Benjelloun, O., Milo, T., 2008. The Active XML project: an overview. *The VLDB Journal*, 17(5), 1019-1040.

Kim, D., and Shen, W., 2007. An Approach to Evaluating Structural Pattern Conformance of UML Models. In *SAC'07, ACM Symposium on Applied Computing*, 1404-1408, ACM Press.

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

Peltz, C., 2003. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10), 46-52.

Thies, G., Vossen, G., 2009. Modelling Web-Oriented Architectures. In *APCCM'09, Asia-Pacific Conference on Conceptual Modelling*. Australian Computer Society Press.

Tolk, A., 2006. What comes after the Semantic Web - PADS Implications for the Dynamic Web. In *PADS'06, 20th Workshop on Principles of Advanced and Distributed Simulation*. IEEE-CS Press.

Zdun, U., Hentrich, C., and van der Aalst, W., 2006. A survey of patterns for Service-Oriented Architectures. *International Journal of Internet Protocol Technology*, 1(3), 132-143.