

A PARTIAL SOLUTION TO THE SEMANTIC WEB SERVICES CHALLENGE PROBLEM USING SWASHUP

The Ruby on Rails Services Mashup Approach

E. Michael Maximilien
IBM Almaden Research Center
650 Harry Road, San Jose, CA, USA
maxim@us.ibm.com

Keywords: Software Engineering, Web Services, B2B Integration, Service Composition, Service Mashups, Web 2.0, Semantic Web Services

Abstract: The Swashup approach to the Semantic Web Services Challenge is primarily a software engineering approach. In particular, our approach heavily leverages the Ruby language, it's Rails framework, and the ability to define languages within the language or domain-specific languages (DSLs). We start by modeling the RosettaNet Pip3A4 messages in such a way that the messages' XSD types directly map to Ruby classes annotated using a DSL. Secondly, we use Ruby to define process mediations that make use of the data elements as plain Ruby objects. Finally, we take advantage of the Rails framework to access SOAP services and to define network endpoints. We demonstrate the effectiveness of our approach by showing how our solution to the level one of first mediation problem is achieved with a small amount of code. We also provide a brief comparison of our approach's main aspects with the other challenge's entries: WSMO/WSMX, WebML, and jABC.

1 INTRODUCTION

The Semantic Web Services Challenge (SWS-Challenge) ¹ is a structured experiment designed to test the applicability of semantic technology (or other technologies) to help resolve interoperability issues with services on the Web.

To concretize Web services interoperability issues and demonstrate how the participants can show how their technologies will solve resulting problems, the challenge presents two classes of problems. First, a set of mediation problems focusing on testing the ability in dealing with complex data mediations as well as simple and more complicated process mediations. Secondly, a series of advanced scenarios focusing on the dynamic discovery of service partners and dynamic integration of their capabilities.

In this paper we present a partial solution to the SWS-Challenge. Specifically, we address the first level of the first mediation scenario. However, we believe that our solution positions us to create the necessary scaffolding and platform to enable us to

address additional scenarios in the future. The approach makes use of modern software engineering techniques which aims primarily at facilitating a programmer in solving the scenarios' problems quickly and efficiently.

1.1 Organization

The remaining of this paper is organized as follows. Section 2 gives some background as well as an overview of our architecture. Section 3 details our solution, including the models for data and for services, as well as algorithms' sketches solving level one of the first mediation problem. In Section 4 we give a brief overview of three other entries to the challenge and briefly compare and contrast their approach with ours. Section 5 is a discussion of our thoughts on our solution and in particular on the importance of the right language, framework, and how using an agile development approach can all help solve the semantic programming problems posed by the challenge. Finally, Section 6 concludes with directions for future work.

¹<http://sws-challenge.org>

2 ARCHITECTURE

Our architecture has its basis on the well-known Model-View-Controller (MVC) (Reenskaug, 1979; Clements et al., 2003) architectural pattern. Our architecture is inherited from the framework we have selected for designing and building our solution: Ruby on Rails².

2.1 Background

The Ruby on Rails (RoR) framework enables agile development of Web applications. The framework contains primitives to help efficiently implement all aspects of an MVC Web application. In particular, there are facilities to allow controllers to invoke external Web services and to model and access data.

The mediator service in our solution is a RoR Web application. This means that the mediator takes advantage of RoR to access remote services, to expose the mediator service endpoint, to manipulate models, and to expose user views. Figure 1 illustrates the high-level architecture of our approach, highlighting the different nodes involved.

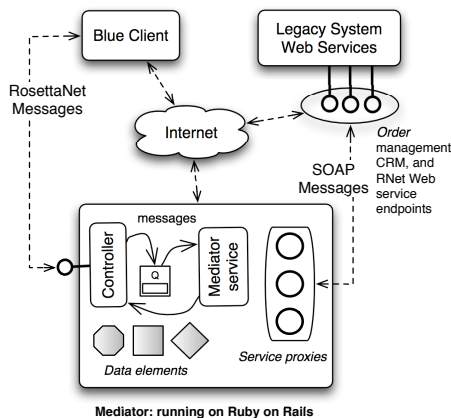


Figure 1: High-level architecture.

While RoR makes it easy to create Web applications with access to external REST and SOAP services, it lacks supports for more advance SOAP support, e.g., document-literal encoded services. Additionally, RoR lacks provisions for easily manipulating complex XML data (beyond parsing) and for some process mediations such as what is required to solve the challenge's scenarios. Our Swashup solution complements the RoR framework by addressing these deficiencies in the context of solving the first mediation scenario.

²<http://rubyonrails.org>

2.2 Overview

The architecture of our solution is driven by our choice of framework and our software engineering approach. In a nutshell, we create a Web application using the RoR framework complemented with other Ruby libraries. Since this Web application essentially composes the data and processes from the *Blue Client's* RosettaNet service and the *Legacy System* back-end services, we are in effect creating a service *mashup*.

As an overview, the architecture includes the following components:

- *Models* which are Ruby classes representing the schema used by the services. These model classes use the Ruby Object to XML mapping³ domain-specific language (DSL) which results in a one-to-one mapping of the class' objects to the data elements of the messages. In many ways, this approach allows the code and data to be one.
- *Mediator Service*, or simply the mediator, is the service component that receives *Pip3A4PurchaseOrderRequest* messages, enqueues them, and processes each by mediating with the *Legacy System* Web services. Additionally the mediator also includes:
 - *Service Proxies* which are object proxies to the external services, to allow remote operations to be invoked by making a local method call. The proxies' method invocations result in the marshaling and unmarshaling of data elements between the mediator and the remote services.
 - *Endpoint Callbacks* which is the network endpoint where the *Blue Client* is able to exchange RosettaNet messages with the mediator.
 - *Data Mediation* which are procedures where mapping of data elements are achieved.
- *Views* which are dynamic Web pages to interface with the end user (when and where necessary).

We believe that the RoR framework extended with some capabilities makes it easy to address all of the elements of the architecture shown in Figure 1.

3 SOLUTION

We give a detail overview of our design and implementation. In particular we focus our attention on four primary aspects:

³<http://roxml.rubyforge.org>

1. *Models* where we describe how we model the various data elements forming the simplified RosettaNet Pip3A4 messages.
2. *Processes* where we give a detail overview of the algorithms that we use to solve the process mediation between the *Blue Client* RosettaNet service and the *Legacy System* services.
3. *Mashups* where we describe how we combine the data model and process mediations to create our solution to the first mediation challenge problem.
4. *Semantics* where we start to make the case for using DSLs in order to add semantics to language elements rather than external descriptions. We also briefly discuss some of the disadvantages of this approach.

3.1 Models

The *Blue Client* asynchronously sends RosettaNet Pi3A4 purchase order request messages to the mediator. The Pip3A4 messages are well specified in a series of XSD schemas which includes all information necessary to enable the mediator to start its message processing.

A large part of the mediator is parsing, representing, and processing the Pip3A4 messages. In our solution we achieve these duties using the ROXML DSL. ROXML allows us to represent each element of the Pip3A4 schema as a Ruby class annotated with the necessary XML attributes, information, as well as structure. Listing 1 shows a generic example of how an XML element with name *elementName* and XSD type *ElementName* is modeled in ROXML.

Listing 1: ROXML models of data elements.

```

1 class ElementName
  include ROXML, Swashup::Creatable
3  xml_name 'elementName'
  xml_attribute :attr_name
5  xml_text :text_element_name
  xml_object :other_name, OtherType
7 end

```

Using the ROXML DSL we describe all of the RosettaNet schemas' elements. Figure 2 illustrates these ROXML classes as a UML class diagram. The `<<roxml>>` stereotype is used to flag the UML class as being a ROXML described class.

Listing 2 shows the actual complete Ruby code for the *Pip3a4PurchaseOrderRequest* ROXML element. It's worth noting that the *xmlObject* DSL call (e.g., line 9) introduces another element, *fromRole*, into this one. Also, the ROXML DSL allows various

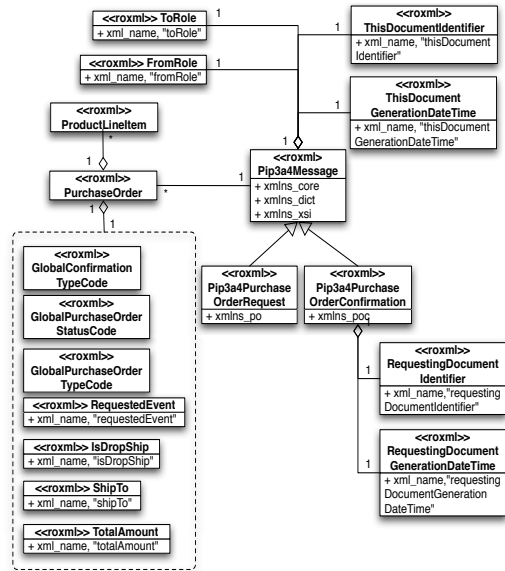


Figure 2: UML model of RosettaNet Pip3A4 messages' main data element.

flexibility in naming the element name, adding attributes, as well as for elements that can be repeated.

Listing 2: Pip3a4PurchaseOrderRequest model.

```

1 class Pip3a4PurchaseOrderRequest
  include ROXML, Swashup::Creatable
3  xml_name 'po:Pip3a4PurchaseOrderRequest'
5  xml_attribute :xmlns_po, 'xmlns:po'
  xml_attribute :xmlns_core, 'xmlns:core'
7  xml_attribute :xmlns_dict, 'xmlns:dict'
9  xml_object :fromRole, FromRole
  xml_object :PurchaseOrder, PurchaseOrder
11  xml_object :thisDocumentGenerationDateTime,
             ThisDocumentGenerationDateTime
13  xml_object :thisDocumentIdentifier,
             ThisDocumentIdentifier
15  xml_object :toRole, ToRole
end

```

The ROXML DSL and library plugin to RoR use the high-level descriptions to generate code that can parse and validate XML documents with the appropriate structures. Similarly Ruby objects created from ROXML annotated classes can also generating appropriate valid XML documents. We use this latter process to do data mediation by manipulating Ruby objects and generate the confirmation message for a purchase order request.

3.2 Processes

The other important job of the challenge’s mediator is to resolve process interactions between the *Blue Client* and the back-end *Legacy System* services. In many ways this can be seen as resolving the protocol between the *Blue Client* and the *Legacy System* services (Singh and Huhns, 2005). In the literature, this is also termed service choreography⁴ or service process composition (Curbera et al., 2002).

We use the RoR framework coupled with our modeling of the RosettaNet Pip3A4 messages to help us mediate the processes in the challenge. Since we cannot assume any synchronization between the different processes involved in the choreography, we add a queue to the mediator’s controller that decouples the input messages from the *Blue Client* and the processing of the messages by the mediator service. Figure 3 shows a UML class diagram of the main components used by the mediator service.

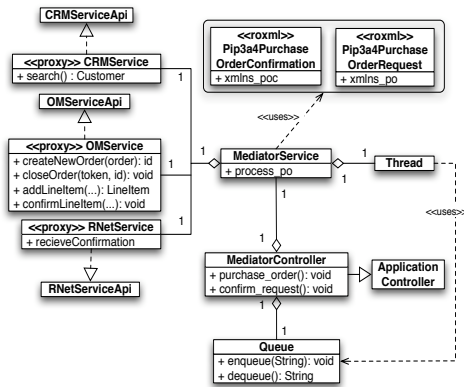


Figure 3: UML model of mediator service.

Figure 4 shows the dynamic interplay of the different components used to design and implement the mediator service. Messages sequenced 1 to 3 show what happens when, asynchronously, a new Pip3A4 purchase order request message is submitted by the *Blue Client*. The message is validated, parsed, and added to a shared queue. The mediator service has a thread that monitors this queue and processes each message until the queue is emptied. Sequences 4, 5, and 6 show this processing. After each purchase order is processed, the order information (e.g., order ID, line items, and so on) are kept in a hash (or dictionary) in the mediator service.

Asynchronously, the mediator controller receives order line item confirmations (e.g., sequences 7 and 8) which results in updates to the appropriate order info record in the mediator’s hash. When the

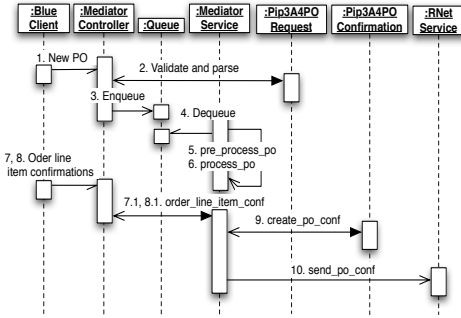


Figure 4: UML sequence diagram for the PO process.

last line item confirmation is received the mediator service’s *post_process_po* method (line 12 of Listing 3) is called which results in the creation of a *Pip3A4OrderConfirmation* with information from the original request message and information from the process mediation. The confirmation message is submitted using the *RNetService* by the *submit_conf* call in line 15. Listing 3 shows abbreviated other parts of the purchase order pre-, post-, and processing algorithms’ sketches.

Listing 3: Algorithms’ sketches for challenge’s data and process mediation (level one).

```

class MediatorService
2   include Swashup, ServiceHelper
   # include many other modules
4
   def purchase_order po_request_message
6       request = Pip3a4PurchaseOrderRequest .
           parse (message)
8       pre_process request
       process request
10    end

12   def post_process request
       @order_info = nil
14       @conf = create_conf (request)
       submit_conf @conf
16       @ack_conf = create_ack_conf (@conf)
       [ @conf, @ack_conf ]
18   end

20   private

22   def pre_process request
       @order_info = OrderInfo.new
24       @line_item_infos = []
       validate_request request
26       @ack = create_ack_request (request)
    
```

⁴<http://www.w3.org/TR/ws-chor-model/>

```

28  end
    def process request
30    request.fromRole.
        PartnerRoleDescriptions.
32      each do |desc|
        pd = desc.PartnerDescription
34      customer = search_customer(pd)
        id = create_order(customer, desc)
36      request.PurchaseOrder.
        ProductLineItems.each do |pli|
38      add_line_item(id, pli)
        end
40      @om_service.
        close_order(@auth_token, id)
42    end
  end
44  def search_customer desc
    text = desc.BusinessDescription.
        businessName.FreeFormText
46    @crm_service.search(text)
48  end
50  def create_ack_conf conf
52    omd = OriginalMessageDigest.create
        :Pip3A4PurchaseOrderConfirmation
54    => conf
    nrm = NonRepudiationInformation.create
56    :OriginalMessageDigest => omd
    ReceiptAcknowledgment.create
58    :NonRepudiationInformation
    => nrm
60  end
62  # Various other private methods
  end

```

3.3 Mashup

At its core, our approach is simply a Web application that aggregates multiple services to achieve a new purpose. It's essentially the composition of the *Blue Client* service and the *Legacy System* services. In essence our solution forms what is termed in the Web 2.0 (Tim O'Reilly, 2005) world, a *mashup*.

Conceptually, mashups are new Web applications used for repurposing existing Web resources and services. They include all three aspects of a typical Web application (model-view-controller) except with addi-

tional functionality. For us, a mashup includes three primary components:

1. *Data mediation* which involves converting, transforming, and massaging the data elements from one service to meet the needs of the operations of another. For instance, we had to do data mediations between the Pip3A4 purchase order request messages to extract necessary information to find the customer via the *CRMService.search()* operation.
2. *Process (or protocol) mediation* which is essentially choreographing between the different services to create a new process. In our solution to the challenge, the protocol resolution resolves the simple request-confirmation protocol of the *Blue Client* to the more involved and finer-grained protocols of the *CRMService*, *OMService*, and *RNet-Service* services. Process mediation includes invoking the various service operations, waiting for asynchronous messages, and sending confirmation messages correlated with the request messages using the correct order identification.
3. *User interface customization* which can be used to elicit user information as well as to display intermittent process information to end users. Depending on the domain, the user interface customization can be as simple as an HTML page, to a more complex series of input forms, or an interactive AJAX UI.

While there are no user interfaces required for the challenge, we added a few simple pages to our solution to enable us to monitor the input message queue as well as some of the mediator's processing steps.

3.4 Semantics

It's worth noting that our solution makes no use of external semantic annotation technologies, such as OWL (McGuinness and van Harmelen, 2004) or RDF/RDF-S. We are able to bypass these technologies in our solution due to the fact that our modeling of the data is self-described. This is similar to how data are represented at the language level in programming languages such as Lisp (McCarthy, 1980; Steele Jr. and Gabriel, 2002) and Self (Ungar and Smith, 1987). In other words, we believe that the description of our data elements using the ROXML DSL carries the necessary semantic information and can be extended to address some level of inferencing (if that is necessary). Additional semantics can be achieved with DSL calls such as *same_as* and other Ruby language semantics, e.g., *alias* and *subclassing*, or additional semantic extensions.

Naturally, some of the challenge's more advanced problems, such as the discovery scenarios may require the need for inferencing code. For instance, in order to discover the best provider meeting some criteria over the purchase orders' line items. Since all of our models are implemented in Ruby, one approach to achieve such 'reasoning' code is done by generating meta-programming code with the models. Note that this is not very different than writing reasoning code with generated programming artifacts using typical annotated semantics such as OWL or RDF. However, since we have not solved the discovery challenge scenarios we leave further discussions for future papers.

4 COMPARISON

We now give a brief overview of how our approach compares with the three main other entries to the challenge. Two of these other approaches are based on software engineering (as ours) and the other one is based on pure semantic technology. After a brief overview of each of the other approaches, we focus our comparison on three main categories: model, process, and tooling.

4.1 WSMX, WebML, and jABC

The WSMX/WSMO (Haller et al., 2005) approach is primarily based on semantic Web technologies. In a nutshell, the RosettaNet Pip3A4 messages elements are modeled using WSMO⁵ ontologies. By a process termed *lifting* and *lowering* the messages are converted into instances of the ontologies and back to XML (for outgoing messages). The *Blue Client* and *Legacy System* Web services operations are modeled using WSMO Goals. The process of the mediator is modeled using a variant of an abstract state machine that uses rules over the ontology elements and the goals. The mediator process' execution results in trying to trace the state machine's valid paths by triggering the goals which would lead to the process' expected results.

The jABC⁶ and WebML⁷ solutions are based on model-driven development approach. In both cases the team started by modeling the Pip3A4 data elements using the appropriate tools of their respective platform. The mediator process is designed using appropriate process modeling tools. In both cases the re-

⁵<http://www.wsmo.org>

⁶<http://www.jabc.de>

⁷<http://www.webml.org>

sult is a process model which represents the choreography of the *Blue Client* service with the *Legacy System* services. The tool generates the appropriate code to represent the Pip3A4 data elements and mediator skeleton process code. The mediator process is completed manually. In the WebML case the code generated for the data elements is kept minimal and uses XSLT to make necessary transformation; whereas, in the jABC case, the data elements are represented as Java classes which can be used as part of the mediator process directly. In both cases, knowledge of the SOAP services descriptions (the WSDL files) allows the tooling system to generate appropriate service proxies to enable the creation of the mediator process.

4.2 Model

From the aspect of modeling the Pip3A4 and service data elements, our Swashup approach is completely different than the WSMX/WSMO approach. We do not model data elements as ontology instances but rather model them directly as Ruby classes using the ROXML DSL and generate the appropriate Ruby instances as the XML messages are parsed. The jABC approach is similar to ours but is done statically and for a statically typed language, whereas our approach gains some dynamism from the Ruby language. This means that we can dynamically change the Ruby classes and instances to match mediation needs. The modifications are possible without having to regenerate the classes from the XML or XSD. The WebML approach is also static but since it uses XSLT to make appropriate XML transformation prior to code generation, this gives it some dynamism and potentially minimizes the impact of the generated code changes on client code.

4.3 Process

Our process modeling and generation approach is also different from the WSMX/WSMO approach and closer to the jABC and WebML. This is expected, since the latter are also based on a software engineering approach. The WSMX/WSMO process model uses a set of rules to determine the actual path to the modeled state machine which then results in the mediation execution. In theory, this could enable a level of flexibility and also of decoupling from the services since the mediated process uses the service operations as goals versus making bound calls to service proxies. However, due to some technical difficulties, the versatility and flexibility of the WSMX/WSMO solution was not tested for this mediation phase so therefore

does not show advantages over the other software engineering approaches.

Our approach to process mediation is similar to the jABC and WebML approaches in that the mediator process is built using the platform's language, libraries, and services. A key difference is that the jABC and WebML have process modeling tools which allows some level of graphical representation of the process prior to code generation. Additionally, our approach is not based on any formalized process model but instead allows us to use the Ruby language and RoR framework directly. A well known BPM language may be beneficial for automatically monitoring part of the process as well as for reuse and potentially difficult issues such as liveness, deadlock, and transactions. However, these aspects are advantages only if the underlying process engine execution supports such features and the use case requires them. Additionally, they may also introduce unnecessary complexity not required for the use case at hand.

4.4 Tooling

Another key aspect where the jABC and WebML approaches differ from the Swashup and WSMX/WSMO approaches is in the level of tooling support they provide. The Swashup approach tooling is made of any IDE or editor supporting the Ruby language and the RoR framework. For our development we primarily used the TextMate⁸ text editor and the Firefox⁹ Web browser. This bare-bone tooling approach is primitive compared to the more advanced graphical tooling of the jABC and WebML approaches. The WSMX/WSMO solution has more tooling support but they are not as mature and thorough as the jABC and WebML toolsets.

5 DISCUSSION

The SWS-Challenge is a combination of real-world enterprise Web-based integration scenarios combined with an implemented test-bed using a modern software platform. Additionally, since the challenge's scenarios are gradual, another aspect of the challenge is to measure the relative easiness or painfulness (mainly based on code changes) in order to address more complicated scenarios.

In this paper we presented a partial solution to the challenge. The solution is partial since it only addresses the first scenario and its first level. However,

⁸<http://www.macromates.com>

⁹<http://www.mozilla.com>

an interesting characteristic of our solution is that it differs from other typical SWS solutions. In particular, our solution makes use and leverages a modern object-oriented language: Ruby, as well as a Ruby-based framework used to create modern Web-based applications: Ruby on Rails.

5.1 Ruby and DSLs

Our solution makes use of two primary features of the Ruby language. Its dynamic typing nature and the ability to define, within Ruby, domain-specific languages (DSLs) that can help abstract concepts in a domain and facilitate their programming.

5.1.1 Dynamic Typing

Ruby is a dynamically typed object-oriented scripting language. This means that objects in the language can be declared and used without first defining their types. This flexibility enables a wide-range of rapid prototyping and reuse features across ones design and algorithms when using Ruby as the implementation language.

For instance, the challenge's messages exchanged between the *Blue Client* and the mediator service (*Pip3A4* purchase order request and confirmation messages) share many common sub-parts, e.g., *core:fromRole*, *core:PurchaseOrder*, and *dict:GlobalPurchaseOrderTypeCode*. Figure 2 illustrates these messages and data elements. The shared elements have almost identical schemas, except for some minor differences. For example, in the confirmation message, the namespace changes, some elements are optional, and also an additional *dict:GlobalPurchaseOrderStatusCode* is added to the *core:PurchaseOrder*. With Ruby's dynamic features, we can model both messages with the same common superclass and at runtime dynamically change the class and its objects to add the necessary missing elements.

One main consequence of this dynamic nature of the language is that the resulting design and code are closely mapped to the data and can change with the data's context. We take full advantage of that feature in our definition of the challenge's schemas and service data types. Ruby's dynamism is also reflected in the mediation procedures which are reduced to only the parts that deal with the required changes between the messages involved. For instance, after receiving a purchase order request and after processing it (i.e., finding the customer, creating the order, adding the order's line items, closing the order, and receiving line item confirmation), our mediator reuses the

same request message object and dynamically modifies it to only account for the differences when submitting the *Pip3A4PurchaseOrderConfirmation* message. The resulting code is focused and more importantly implicitly carries the necessary semantic information.

5.1.2 Domain-Specific Languages

Domain-Specific Languages are ‘mini’ languages built on top of the primary language (Ruby in our case) that help abstract aspects of a particular domain. Using DSLs results in higher-level expressions that are compact, easier to code, and easier to program with. In other words, DSLs allow meta-level programming (Rosenberg, 2007) or also what some now call “intentional programming” (Simonyi et al., 2006). In many ways, the DSLs allow some of the semantics of the domain to be modeled in the resulting code.

In particular, Ruby’s DSL support results in code whose syntax is compact and legible. For instance, we use the ROXML DSL to represent all of the challenge’s messages’ elements and sub-elements. Examples of the ROXML DSL are shown in Section 3.1. Listing 4 shows the complete definition of the element *PurchaseOrder* which is part of both the RosettaNet request and confirmation messages.

Listing 4: PurchaseOrder ROXML class.

```

1 class PurchaseOrder
  include ROXML, Swashup::Creatable
3  xml_name 'PurchaseOrder'
  xml_text :GlobalConfirmationTypeCode ,
5    'dict:GlobalConfirmationTypeCode'
  xml_text :GlobalPurchaseOrderStatusCode ,
7    'dict:GlobalPurchaseOrderStatusCode'
  xml_text :GlobalPurchaseOrderTypeCode ,
9    'dict:GlobalPurchaseOrderTypeCode'

11  xml_object :isDropShip , IsDropShip
  xml_object :ProductLineItem ,
13    ProductLineItem
  xml_object :requestedEvent ,
15    RequestedEvent
  xml_object :shipTo , ShipTo
17  xml_object :totalAmount , TotalAmount
end

```

The statement at line 3 defines the name for the expected element, and the statements on lines 5 and 7 indicate that the *PurchaseOrder* element is composed of XML simple types and other complex types defined as other Ruby classes.

Using the ROXML DSL to define the different messages and their elements means that the mediator can treat both, XML and the necessary Ruby objects, to perform its tasks in a uniform fashion. This results in simple, intentional code, representing its intent and semantics implicitly. Section 3.2 illustrated this point in more details by showing examples of the mediator process’ mediations.

5.2 Framework

Another aspect of our approach is in the leveraging of the RoR framework. In particular we take advantage of the various libraries and DSLs in RoR to consume Web services, to expose an endpoint for external consumptions, to create view pages that aggregates data, and to create unit and functional tests.

In Section 3.1 we mentioned how we used RoR’s facilities to quickly create proxies to external SOAP Web services. We had to expand the library to support SOAP document-literal encoding which is the style used in the challenge’s *Legacy System* Web services. The resulting proxy objects enable us to code our process mediation as if we were creating a process amongst local objects.

We use RoR’s concept of a Web application controller to define our SOAP Web service endpoint which receives *Pip3A4PurchaseOrderRequest* and sends and receives the acknowledgments and confirmation messages. In RoR the endpoint is essentially a method defined in the controller class which is called whenever an HTTP request (PUT, GET, POST, or DELETE) is received. The request’s URL includes the controller’s name and method in it’s path. Using our ROXML modeling of the expected messages, we parse the SOAP messages to extract the payload and parse it to create the corresponding Pip3A4 message elements objects. Afterward, the request message object is simply enqueued for future processing by the mediator.

5.3 Agility

The final feature of the RoR framework that we use is in the support for Agile development and design. In particular for our solution agility comes in the ability to create and manage unit and functional tests. Our solution comprises a fair amount of unit and functional tests, especially since such artifacts are usually not done for research development or for challenges. Table 1 summarizes the source and test lines of code (LOC), classes, and methods statistics for our solution.

Table 1: Source and tests code statistics. Total code LOC is 1996 and total test LOC is 793 for a code-to-test ratio of 1 : 0.4

Name	LOC	Classes	Methods
Controllers	59	6	12
Helpers	1331	88	98
Models	380	41	4
APIs	214	2	30
Functional tests	390	9	30
Unit tests	403	2	10
<i>Totals</i>	2789	149	184

We develop these tests to follow the agile practice of Test-Driven Development (Maximilien and Williams, 2003). A practical and important use of these tests is in allowing us to gradually model and test our solution and also to help curb runtime errors that can plague dynamic languages and systems. The practice of such Agile modeling greatly helps when we are iteratively modeling the domain, i.e., the RosettaNet schemas, the solution’s data and process mediations.

6 CONCLUSION

Our solution to the SWS-Challenge is a partial solution. This is due to the simple fact that we only had time and resources to address the first part of the mediation scenarios. Our intentions is to complete the solution by addressing the remaining scenarios. Additionally, we also intend to measure the required changes to the code and platform as we move to the more complex scenarios. We are keeping various statistics on the platform and will share them freely with the challenge’s organizers. Finally, since we are proposing that our software engineering based approach can help in resolving the challenges’ other problems, additional and thorough comparisons with the other challenges’ participants will help give insights into the relative merits of our approach.

ACKNOWLEDGEMENTS

I wish to acknowledge N. C. Narendra from IBM India Research Lab and David Martin from SRI International for various discussions on earlier versions of the Swashup project as well as discussions about the SWS-Challenge problem scenarios. I wish to also thank Holger Lausen, Michal Zaremba, and Zhixian Yan from DERI, as well as Charles Petrie from Stanford University, for their support on using the SWS-

Challenge platform, clarifying problem descriptions, and for answering questions when difficulties arose.

REFERENCES

- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2003). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA.
- Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., and Weerawarana, S. (2002). Business Process Execution Language for Web Services, Version 1.0. www-128.ibm.com/developerworks/library/specification/ws-bpel/.
- Haller, A., Cimpian, E., Mocan, A., Oren, E., and Bussler, C. (2005). WSMX - A Semantic Service-Oriented Architecture. In *Proceedings of the 3rd IEEE International Conference on Web Services*, pages 321–328, Las Vegas, NV.
- Maximilien, E. M. and Williams, L. (2003). Assessing Test-Driven Development at IBM. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 564–569, Portland, OR. IEEE Computer Society.
- McCarthy, J. (1980). Lisp—Notes on its Past and Future—1980. www-formal.stanford.edu/jmc/lisp20th.pdf.
- McGuinness, D. L. and van Harmelen, F. (2004). OWL Web Ontology Language Overview. <http://www.w3.org/TR/owl-features/>. W3C recommendation.
- Reenskaug, T. (1979). Models-Views-Controllers. Technical report, XEROX Palo Alto Research Center, Palo Alto, CA.
- Rosenberg, S. (2007). Anything You Can Do, I Can Do Meta. *MIT Technology Review*, pages 36–48.
- Simonyi, C., Christerson, M., and Clifford, S. (2006). Intentional software. In *Proceedings of the ACM conference on Object-Oriented Programming Systems Languages Applications (OOPSLA’06)*, pages 451–464, Portland, OR.
- Singh, M. P. and Huhns, M. N. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, Chichester, UK.
- Steele Jr., G. L. and Gabriel, R. P. (2002). The Evolution of Lisp. www.dreamsongs.com/NewFiles/HOPL2-Uncut.pdf.
- Tim O’Reilly (2005). What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. www.oreillynet.com/lpt/a/6228.
- Ungar, D. and Smith, R. B. (1987). Self: The Power of Simplicity. In *Proceedings of the ACM conference on Object-Oriented Programming Systems Languages Applications (OOPSLA’87)*, pages 227–241, Orlando, FL.