Approved for Public Release; Distribution Unlimited
Case # 06-0814

Using Data Semantics to Enable Automatic Composition of Web Services

Danny Gagne, Marwan Sabbouh, Dr., and Scott Bennett, Susan Powers

***Abstract*— This paper demonstrates the automatic creation of a web service that chains together existing web services to achieve a particular goal. The generated service implements the necessary workflows to convert an instance data of one system into an instance data of another. This paper further demonstrates the reconciliation of structural, syntactic, and representational mismatches between the input instance and the desired output instance.**

I. Introduction

Traditional techniques for enterprise integration require the manual implementation of workflows in the form of code or Web Services Business Process Execution Language (WSBPEL) [1]. We developed a tool that automatically generates the necessary integration code in the form of a web service. This web service enables information originating in one information system to be reused in another information system. The code generator leverages a small set of RDF/OWL mapping relations [2,3] in conjunction with context ontologies, ubiquitous enterprise concepts such as Types of Things, Time, and Position (What, When and Where) that relate all the various representations across the enterprise [2]. The generated web service implements a workflow that typically consists of the consecutive invocation of web services. These web services perform functions such as retrieval of data from data sources, translation between various representations of time and position, and other similar operations.

To further illustrate the working of the code generator, we will begin by discussing the integration of two air flight systems: Air Mobility (AM) and the Air Operations (AO). First a little background, the AM system is responsible for many different types of missions including: mid-air refueling, the movement of vehicles, and Air Force One. The AO system is primarily concerned with offensive and defensive missions. These were developed independently and built for the specific needs of the users.

In the following scenario we would like to integrate these two systems so that the AO system can be kept aware of all of the AM missions. To this end a third system will need to be created to accomplish this integration. It will be a web service that implements the necessary workflows to convert an AM instance into an AO instance. The code generator is then able to create this web service provided that each system is described with an IS ontology, representing the context [4] of each system, which is mapped to the context ontologies, and has the necessary mappings between them.

To keep things simple, assume that the only information that the AO system needs from the AM system is the plane's call sign, its aircraft type, and its position; however, this information will need to be structured in such a way that the AO system can process it natively. This will allow the AO system to display the correct airplane icon and call sign in the right location. So the generated web service will need to retrieve an instance of an AM mission and convert the data to an instance of an AO mission, in doing so it will need to overcome structural, representational, and semantic mismatches.

To achieve this end, the code generator creates an empty AO instance which is then populated with data from the ontologies and the AM instance. There are three types of transformations that can occur when moving data from one system to another. This simplest is that of merely copying the data to the correct locations to harmonize structural differences. In AM-AO example, the call sign needs to be transferred as it is so that a user in the AO system can contact the plane as required. A slightly more complicated transformation is needed to correct syntactic differences when dealing with enumerated data types. This is accomplished by substitution, in our case every time an "A10A" is encountered it is replaced by "AO10A", and then it is transferred to its correct position in the AO instance.

The third and final transformation involves the reconciliation of representational mismatches; that is, cases in which the data needs to be converted from one representation to another. The code is generated in two steps; the first step converts the data into the desired representation, and the second inserts this data into the correct position. This requires an understanding of the context the data has, for example if a measurement is in miles or kilometers [reference to the space probe accident], or the time zone of a time. To convert an AM coordstring, which is a coordinate stored in UTM format (for example: "21 N 678076 5423265"), to an AO coordinate, which has a geodetic format (an xml representation that has

both latitude and longitude), requires the invocation a third party web service called Geotrans. Geotrans is mapped into the position [5] context ontology, and is capable of converting between different representations of location data. After Geotrans is executed it returns an instance containing latitude and longitude that is moved to the AO instance.

The next section will briefly describe the methodology that is used to create the ontologies and map them together, and then we will describe the code generation process in detail.

II. BACKGROUND

The system ontologies, expressed in OWL\RDF [6, 7], define the local semantics for each system, and are enriched through their relations to the various context ontologies. Furthermore, through their mapping to other system ontologies, as shown in Fig. 1, they enable the code generator to reason over the mapped ontology to reconcile syntactic, structural, and representational mismatches.

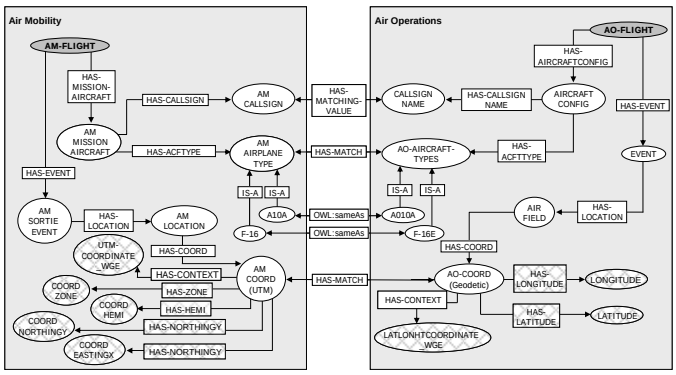

Fig.1. Mappings between the AM and AO System Ontologies.

This diagram, Fig. 1, highlights the following four relationships: *HAS-MATCHING-VALUE, HAS-CONTEXT, HAS-MATCH, and OWL:sameAs.* These relationships are created when integrating the two systems; each system has already been mapped to the appropriate context ontologies. The *HAS-MATCHING-VALUE* relation signifies that the values are equivalent, whereas *HAS-MATCH* only denotes semantic correspondence. That is, concepts that are connected by *HAS-MATCH* relationships must also share either a context (*HAS-CONTEXT*), or have instances that are linked together using *OWL:sameAs*. The sharing of a context permits the reconciliation of representational mismatches between two concepts.

The system ontologies are not complete until they have been mapped to their respective web services. This is accomplished by translating the Web Service Description Language (WSDL) [8] file for each web service into its own WSDL ontology, and web service upper ontology [9]. The web service upper ontology is mapped to the system ontology, and serves a similar purpose as that of the OWL-S Service Profile [10]. The WSDL ontology contains all of the information in a WSDL file, and is mapped to the system ontology to annotate the XML elements and attributes, defined in the WSDL file, with semantics. This allows for a concept in the system ontology to be shared between multiple web services. For instance, a concept in the system ontology can be mapped to the input of one web service and to the output of another web service. This information plays a role when the output of one web service is used as the input to another web service.

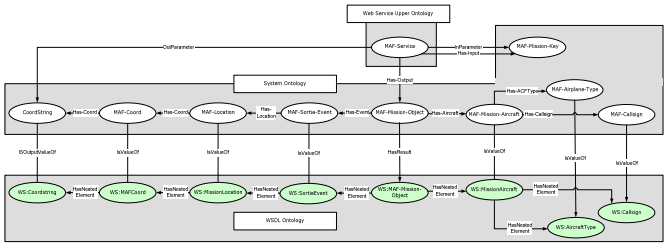

Fig. 2. Integration of the MAF-SERVICE to the AM ontology.

Fig. 2 illustrates the mappings that are created when adding a web service to the AM system ontology. The web service upper ontology, MAF-SERVICE, is mapped to concepts in the system ontology using the OWL/RDF mapping relations: *HAS-RESULT, HAS-INPUT, HAS-OUTPUT, IN-PARAMETER,* and *OUT-PARAMETER*. These same concepts in the system ontology are mapped to the WSDL ontology using the OWL/RDF mapping relation *IS-VALUE-OF*. For example, COORDSTRING, the *OUT-PARAMETER* of the MAF-SERVICE as shown in Fig. 2 is also linked to WS:COORDSTRING using the OWL/RDF mapping relation *IsOutputValueOf*.

Once these mappings have been completed, we reason over the ontologies using either a Direct Path Query (DPQ) or an Incoming Intersection Query (IIQ) to discover workflows, to retrieve default parameters, to reconcile structural, syntactic, and representation mismatches. Both the DPQ and IIQ algorithms make use of RDF/OWL inferences and are described as follows: the DPQ is given a list of input values and a desired output, the DPQ creates the set of all the direct paths that lead to the desired output concept from the input concepts. The IIQ is also given a list of input values and a desired output and performs a series of DPQs in the following manner. First, the algorithm uses a DPQ to create the set of all the direct paths, starting from any node in the graph, that lead to the desired output concept. Second, for each input value, the algorithm uses a DPQ to create the set of direct paths, starting from any node in the graph, that lead to that input value. Third, the algorithm returns the intersection of these sets. These algorithms are provided to the code generator as part of the semantic framework [2, 9]. In the next section we discuss how the code generator uses these queries to produce the code.

III. CODE GENERATION

To facilitate the integration of each system we generate a web service that can convert an instance from one IS to another. This is accomplished by reasoning over the ontologies presented in the previous section. There are three distinct phases that define the creation of the service. The first is the processing of the high level workflow, the second is the processing of the nested parallel workflows, and the third is the creation of the output instance. Once these phases have been completed the web service is compiled and deployed to the web server.

The code generator uses a DPQ with the input value of MAF-MISSION-KEY and the output value of CAF-AIRMISSION to retrieve the high level workflow shown in Fig. 3. This workflow denotes the steps that are used to retrieve the data, and if necessary the additional steps needed to reconcile mismatches to create the correct instance for the output. In our case Fig. 3 specifies that the MAF-MISSIONKEY, a mission id, will be used to retrieve a MAF-MISSION-OBJECT from the MAF-SERVICE web service. We refer to the MAF-MISION-KEY as the input object to the webservice and the MAF-MISSION-OBJECT as the output object. The MAF-MISSION-OBJECT will then be used as the input to the ONTOMAPPER process; the ONTOMAPPER concept signifies that there are structural, syntactic, and representational mismatches between the AM and the AO system ontologies that need to be reconciled. Whenever ONTOMAPPER is encountered, the generator uses a DPQ to retrieve the parallel workflows that are needed to resolve these mismatches.

Fig. 3. High level workflow to convert a MAF-MISSION-KEY to a CAF-AIRMISSION

The processing of the workflows will be described in the next section.

A. Processing The Workflow

To convert the workflow from an RDF/XML document into C# [11] code, the workflow is loaded into memory as an RDF graph. Then the full class definition of each concept is retrieved using services provided by the semantic framework. This data is then merged into the RDF graph. Next each concept is assigned an implementation type determined by its position in the graph. The graph of implementation types provides an environment from which code is generated. The implementation types denote whether a concept represents a web service, complex type, complex array type, simple type, simple array type, polymorphic type, composite type, reference name, method name, instance value, parameter type, onto web service, or a mock type. For brevity we will discuss those types representing web services, parameter types, complex types, composite types.

A concept is assigned the web service implementation type if the concept is a subclass of the WEBSERVICE concept. The type representing the web service is defined as having the following members: references to the composite types denoting the input and output objects, arrays of the parameter implementation types denoting the [in]parameters and [out] parameters, a string containing the WSDL, a CodeNamespace[12] containing the proxy code for the webservice, and the namespace, class, and method name used to invoke this proxy.

Concepts in the graph that are the object of the relationships *IN-PARAMETER* and *OUT-PARAMETER* are assigned the parameter implementation type. It is defined as having the following members: a reference to the implementation type of the object of *IS-VALUE-Of* relationship (shown in Fig. 2), a reference to the implementation type of the object of *IS-OUPUT-VALUE-OF* relationship, its type as declared in the xml schema [13] (int, string, double, etc), and references to the default values of the [in]parameter and the [out]parameter.

A complex implementation type is associated with a concept in the WSDL ontology which is defined using the relationships *HAS-NESTED-ELEMENTS* and *HAS-ATTRIBUTES*. It is defined as having the following members: namespace name, class name, and an array of implementation types which can be a complex type, complex type array, polymorphic type, simple type, and a simple type array.

A composite implementation type is assigned with a concept that exists in the workflow that is neither a web service type, an onto web service type, nor a parameter type. It is defined as having a reference to another implementation type.

B. Workflow Implementation

Once each concept has been assigned an implementation type, the workflow in Fig. 3 is implemented as a web method, as shown in Fig. 4. To generate the WebMethod, the code generator creates an environment stack that stores variable name/implementation type pairs that will be declared in this method's scope.

```
[WebMethod()]
public virtual NS2.CAFAirMission Execute12(string var13)
{
        NS3.MAFService service14 = new NS3.MAFService();
        NS3.MAFMissionObject var15 = service14.GetMAFMission(var13);
        NS2.CAFAirMission var18 = this.ConvertObject10(var15);
        return var18;
}
```

Fig.4. High level workflow translated to C# code.

The code generator processes the workflow step by step. In this case, the processing of MAF-MISSION-KEY results in the creation of a variable (var13) which is placed into the environment stack with the MAF-MISSION-KEY concept's implementation type. Note that the implementation type also contains the type information of the variable, in this case type string. The code generator then retrieves the next item in the workflow that is of type web service or onto web service. In this case the code generator processes the MAF-SERVICE by emitting the code to instantiate its web service proxy (e.g. NS3.MAFService service14 = new NS3.MAFService();) and stores the variable name/implementation (service14/MAF-SERVICE) type on the stack.

Before the code to invoke the web service can be emitted the code generator retrieves the parameter implementation types from the web service implementation type, and checks to see if any of these parameter types exist in the environment stack. If so, the variable name associated with the implementation type is retrieved, in this case var13. Once all variable names are found, they are passed as arguments in the method invocation. In certain situations a web service may require input arguments that are not provided by the high level workflow, as a result, they are not part of the environment stack. In our approach the semantic framework offers facilities to associate default values with certain parameters and to retrieve these values which are then used as arguments in the method invocation. If a default value for the parameter cannot be ascertained then a null value is used. Next the code generator retrieves from the web service implementation type the output implementation type. It then creates a variable (var15) to store the return value of the service, and places variable and associated implementation type on the environment stack (e.g. var15\MAF-MISSION-OBJECT). This allows the following line of code to be emitted: NS3.MAFMissionObject var15 = service14.GetMAFMission(var13);

C. Reconciliation of Structural, Syntactic, and Representational Mismatches

The code generator then retrieves the next item in the workflow which is ontomapper. Ontomapper is a mediator that translates an input instance to a desired output instance. It first creates an empty instance of the output, and then populates it with data from the input instance. Some of the input instance data can be copied as is to the correct place in the output, reconciling structural mismatches only, other input data will need have a substitution preformed to also resolve syntactic mismatches. Further still some data will need to be acted upon by workflows to convert the data into the correct representation.

The code generator reconciles only structural mismatches when concepts are mapped using HAS-MATCHING-VALUE (Fig. 1). The code generator reconciles structural and syntactic mismatches when concepts are mapped using HAS-MATCH, and instances of the concepts are mapped using owl:sameAs. The code generator reconciles structural and representational mismatches when concepts are mapped using HAS-MATCH and share a context (HAS-CONTEXT). In which case, the code generator retrieves all the parallel workflows necessary to enable the creation of the output instance from the input instance. For example, one workflow reconciles position representation mismatch (UTM to Geodetic), while another reconciles time representation mismatches (HH:MM:SS ET to HH:MM:SS Z), and so on. In this case, there exists a representation mismatch between the MAF-COORD and CAF-COORD concepts, and the workflow to reconcile them is shown in Fig. 5.

Fig. 5. Workflow to resolve representational mismatches derived mapped ontologies.

The workflows are retrieved by using facilities provided by the semantic framework [2].These workflows are processed in a similar way to the high level workflow. What remains to be shown is how the code generator emits the code to generate a new instance.

1) Instance Generation

The code generator emits the code to instantiate an empty CAF-AIRMISSION instance from the WS:CAF-AIRMISSION implementation type. Each mapping relation between system ontologies results in the creation of assignment statements. In which case, the reconciling of structural mismatches is reduced to determining the signature between the left hand side and the right hand side of the statement. Recall, that the WS:CAFAIRMISSION has a complex implementation type which contains an array of its children's implementation types, and each implementation type has reference to its corresponding concept in the system ontology. Next the code generator processes each child's implementation type to determine how its concept is mapped to a concept in the input object, MAF-AIR-MISSION. For each occurrence of the mapping relation between concepts in the output (CAF-AIR-MISSION) and input object, the code generator determines the assignment statement by issuing a DPQ on the input object root element, WS:MAF-MISSION-OBJECT, and the object of the mapping relation to determine the signature of the right hand side of the assignment statement, and by issuing a DPQ on the root element of the of the output object, WS:CAFAIRMISSION, and the subject of the mapping relation to determine the signature of the left hand side of assignment statement.

In the case where the mapping relation is has-matching-value, the code generator then emits the code to perform the assignment between the left hand side and the right hand side, effectively transferring the value of the concept from AM to the AO, for example: returnObject.AircraftConfig.CallSignName = toBeConverted.MissionAircraft.CallSign;

In the case where the mapping relation is *HAS-MATC*H, the code generator must also resolve syntactic mismatches, by using an IIQ to retrieve a list of the instances of the AM concepts and their corresponding instances in the AO. A new method is then generated that substitutes each AM instance by its AO equivalent, as shown in Fig. 6.

```
returnObject.AircraftConfig.AircraftType = this.Swap68(toBeConverted.MissionAircraft.AircraftType);
…

public virtual string Swap68(string toBeConverted)
{
        if (("A10A".ToLower() == toBeConverted.ToLower()))
        {
                return "AO10A";
        }
 …
}
```

Fig. 6. Reconciling Syntactic and Structural Mismatches

In the case where the mapping relation is both *HAS-MATCH* and *HAS-CONTEXT*, the code generator must resolve the representational mismatches. This is done by finding the correct parallel workflow to convert between the instances of the concepts connected by the *HAS-MATCH* relation. This workflow is then processed in same way as the high level workflow. The output structure returned has some of the same members as the output object, i.e. CAF-AIRMISSION. Then the code generator emits the code to assign the data from this structure to the output object (Fig. 7), thus reconciling representational mismatches between the AO and the AM.

```
int count20 = toBeConverted.SortieEvent.Length;
returnObject.CAFEvent = new NS2.CAFEventType[count20];
for (int i21 = 0; (i21 < count20); i21 = (i21 + 1))
{
    …
    NS5.GeoTransService service33 = new NS5.GeoTransService();
    NS5.Geotrans var34 =
        service33.GeoTrans("UTM", "WGE", "GEODETIC", "WGE", null, null, null, null,
        toBeConverted.SortieEvent[i21].MissionLocation.MAFCoord.CoordString);

    returnObject.CAFEvent[i21].ICAOLocation.CAFCoord.Latitude =
        ((NS5.GEODETIC)var34.Item).OutputCoordinateList[0].Latitude;

    returnObject.CAFEvent[i21].ICAOLocation.CAFCoord.Longitude =
        ((NS5.GEODETIC)var34.Item).OutputCoordinateList[0].Longitude;
}
```

Fig. 7 Reconciling representational and structural mismatches.

2) Array Management

The code generator must identify and handle arrays to reconcile structural mismatches between the input and output objects. This results in the code generator emitting a for loop construct. When the code generator finds a mapping relation between the input and output objects, it uses a DPQ between the input object (WS:MAF-MISSION-OBJECT) and the object of the mapping relation and a DQP between the output object(WS:CAFAIRMISSION) and the subject of the mapping relation, resulting in a directed graph for the right and left side, respectively. The code generator then traverses each graph to identify the concepts that are of array types. A concept in the WSDL ontology is determined to be of an array type if its corresponding element in the WSDL file has a maxOccurrs with a value greater than one. The code generator simply associates the first concept of array type on the left hand side with the first concept of array type on the right side when assigning the index value, a reference to the for loop's iterator. This aligns the structures so that objects contained by the objects in the array will be processed using the same iterator. The code generator emits the code to initialize the array denoted by the concept on the left hand side to the length of that on the right hand side. It then emits a for loop construct and stores the iterator variable in an array environment. The code generator uses the iterator variable to generate the correct signature for accessing the array's child nodes, this is shown in Fig. 7. If an array is only detected on one side then its index is set to zero, since there is only room for one corresponding structure on the other side. Once the processing of child nodes is completed the generator removes the iterator information from the environment and the for loop is closed.

Finally, a flow chart describing this entire process of instance generation is shown in Fig. 8.

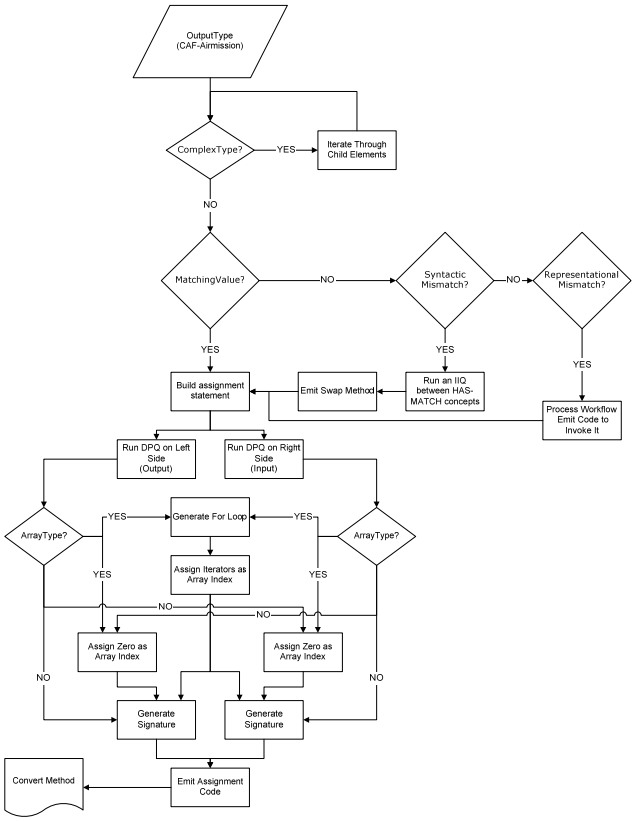

Fig. 8. The process of emitting the code to generate an instance.

IV. Discussion and Relation to the Literature

Compared to Semantic Web Services [14] based solutions, this technique is lightweight in that it builds on current W3C standards (XML Schema, RDF, OWL, WSDL) for building web services, and for specifying semantics, without requiring the many specifications introduced by WSMO [15, 16] and OWL-S [10]. In addition to the small set of RDF/OWL mappings, we only use RDF and OWL to build the IS ontologies, which describe the local semantics of a web service, thus alleviating the need for building expensive domain ontologies [17, 18]. Furthermore, the building of IS ontologies is facilitated by the context ontologies, since concepts defined in the context ontologies are used to define the IS ontologies. It is important to note that the context ontologies are built once for reuse by all integration activities. Once these ontologies are in place, the annotations of existing XML schemas with semantics, and the mappings of the ontologies are straight forward. Another distinct advantage to this approach is that the generated code runs in a typical .Net environment, and the generated code is no different than the code written by a programmer today. Alternatively, we could have generated a WSBPEL process using the same techniques just as easily. Finally, it is worth mentioning that we see our tools as part of an Enterprise Service Bus (ESB) [19], where an architect can use them at design time, in order facilitate the task of code generation.

V. Future Work

Currently, we are in the process of demonstrating the scalability of this approach. Our early results suggest that this mapping technique scales O(n) with respect to the number of systems that exists in the enterprise. This is the result of making some of the RDF\OWL mappings transitive which results in the automatic generation of most of the links between IS ontologies. We are also investigating the global properties [37,38], e.g. path length and clustering coefficient, of the emerging complex network [39] that arise from this mapping technique. For instance, a small path length indicates that the DPQ and IIQ would work well in an enterprise with millions of systems.

References

[1] OASIS Web Services Business Process Execution Language (WSBPEL), http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[2] Sabbouh, M, DeRosa, J.K., Powers S., Bennett S., "Using semantic web technologies to enable interoperability of disparate information systems", Mitre Technical Report MTR05B0000083, 2005, available from: http://www.mitre.org/work/tech_papers/tech_papers_05/05_1025/

[3] Crubezy,M.,Pincus,Z.,Musen.M.A., "Mediating knowledge between application components", Semantic Integration Workshop of the Second International Semantic Web Conference (ISWC-03), Sanibel Island, Florida, CEUR, 82. 2003.

[4] McCarthy, J., GENERALITY IN ARTIFICIAL INTELLIGENCE, Communications of the ACM, 30(12):1030-1035, 1987

[5] Kazura A., Sabbouh M., 2005, Position ontology, Available fromhttp://www.openchannelfoundation.org/orders/index.php?group_id=348, The Open Channel Foundation

[6] Resource Description Framework (RDF), World Wide Web Consortium, http://www.w3.org/rdf/

[7] Web Ontology Language (OWL), World Wide Web Consortium, http://www.w3.org/2004/OWL/

[8] Webservice Description Language (WSDL) 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315, June 2006

[9] Sabbouh, M., DeRosa, J.K., "Using Semantic web technologies to integrate software components", Proceedings of the ISWC 2004 Workshop on Semantic Web Services. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-119/paper8.pdf

[10] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/, November, 2004

[11] C# Language Specification, ECMA-334, http://www.ecma-international.org/publications/standards/Ecma-334.htm, June 2006

[12] CodeNamepsace Class Reference, http://msdn2.microsoft.com/en-us/system.codedom.codenamespace.aspx, June 2006

[13] XML Schema, http://www.w3.org/XML/Schema, June 2006

[14] McIlraith S., Son T.,Zeng H., Semantic Web services. In IEEE Intelligent Systems (Special Issue on the Semantic Web), March/April 2001.

[15] Bruijn, J., Fensel, D., Keller, U., Lara, R., "Using the web service modeling ontology to enable semantic e-business, Communications of the ACM Volume 48, Number 12 (2005), Pages 43-47

[16] Web Service Modeling Ontology (WSMO), http://www.w3.org/Submission/WSMO/, June 2005

[17] T.R. Gruber. "A translation approach to portable ontologies", J on Knowledge Acquisition, Vol 5(2), p199-220, (1993)

[18] Guarino, N, ed. 1998. Formal Ontology in Information Systems. Amsterdam.: IOS Press. Proceedings of the First/ International Conference (FOIS'98), June 6-8, Trento, Italy.

[19] Patterns: Implementing an SOA using an Enterprise Service Bus, Keen, M., Acharya, A., Bishop, S., Hopkins, A., Milinski, S., Nott, C., Robinson, R, Adams, J., Vershueren, P., http://www.redbooks.ibm.com/redbooks/pdfs/sg246346.pdf, July 2004

First A. Author (M'76–SM'81–F'87) and the other authors may include biographies at the end of regular papers. Biographies are often not included in conference-related papers. This author became a Member (M) of IEEE in 1976, a Senior Member (SM) in 1981, and a Fellow (F) in 1987. The first paragraph may contain a place and/or date of birth (list place, then date). Next, the author's educational background is listed. The degrees should be listed with type of degree in what field, which institution, city, state or country, and year degree was earned. The author's major field of study should be lower-cased.

The second paragraph uses the pronoun of the person (he or she) and not the author's last name. It lists military and work experience, including summer and fellowship jobs. Job titles are capitalized. The current job must have a location; previous positions may be listed without one. Information concerning previous publications may be included. Try not to list more than three books or published articles. The format for listing publishers of a book within the biography is: title of book (city, state: publisher name, year) similar to a reference. Current and previous research interests ends the paragraph.

The third paragraph begins with the author's title and last name (e.g., Dr. Smith, Prof. Jones, Mr. Kajor, Ms. Hunter). List any memberships in professional societies other than the IEEE. Finally, list any awards and work for IEEE committees and publications. If a photograph is provided, the biography will be indented around it. The photograph is placed at the top left of the biography. Personal hobbies will be deleted from the biography.

APPRENDIX I: GENERATED CODE

Note: Due to length constraints the first 1000 lines of code generated by wsdl.exe have been removed.

```
namespace NS
{
   using System;
   using System.Diagnostics;
   using System.Xml.Serialization;
   using System.Web;
   using System.Web.Services.Protocols;
   using System.Web.Services;


   public class GeneratedService
   {

      [WebMethod()]
      public virtual NS2.CAFAirMission Execute12(string var13)
      {
         NS3.MAFService service14 = new NS3.MAFService();
         NS3.MAFMissionObject var15 = service14.GetMAFMission(var13);
         NS2.CAFAirMission var18 = this.ConvertObject10(var15);
         return var18;
      }

      public virtual NS2.CAFAirMission
ConvertObject10(NS3.MAFMissionObject toBeConverted)
      {
         NS2.CAFAirMission returnObject = new NS2.CAFAirMission();
         returnObject.AircraftConfig = new NS2.AircraftConfigType();
         returnObject.AircraftConfig.CallSignName =
toBeConverted.MissionAircraft.CallSign;
         returnObject.AircraftConfig.AircraftType =
this.Swap68(toBeConverted.MissionAircraft.AircraftType);

         int count20 = toBeConverted.SortieEvent.Length;
         returnObject.CAFEvent = new NS2.CAFEventType[count20];
         for (int i21 = 0; (i21 < count20); i21 = (i21 + 1))
         {
            returnObject.CAFEvent[i21] = new NS2.CAFEventType();
            returnObject.CAFEvent[i21].EventType =
this.Swap30(toBeConverted.SortieEvent[i21].EventType);
            returnObject.CAFEvent[i21].EventStartTime =
toBeConverted.SortieEvent[i21].EventTime;
            returnObject.CAFEvent[i21].ICAOLocation = new
NS2.ICAOLocationType();
            returnObject.CAFEvent[i21].ICAOLocation.ICAOState =
toBeConverted.SortieEvent[i21].MissionLocation.ICAOCD;
            returnObject.CAFEvent[i21].ICAOLocation.CAFCoord = new
NS2.CAFCoordType();
            NS5.GeoTransService service33 = new NS5.GeoTransService();
            NS5.Geotrans var34 = service33.GeoTrans("UTM", "WGE",
"GEODETIC", "WGE", null, null, null, null,
toBeConverted.SortieEvent[i21].MissionLocation.MAFCoord.CoordString);
            returnObject.CAFEvent[i21].ICAOLocation.CAFCoord.Latitude =
((NS5.GEODETIC)var34.Item).OutputCoordinateList[0].Latitude;
            returnObject.CAFEvent[i21].ICAOLocation.CAFCoord.Longitude
= ((NS5.GEODETIC)var34.Item).OutputCoordinateList[0].Longitude;
         }
         returnObject.TaskUnit = toBeConverted.TaskUnit;
         returnObject.StatusAndResult = new NS2.StatusAndResultType();
         return returnObject;
      }

      public virtual string Swap30(string toBeConverted)
      {
         if (("LANDING".ToLower() == toBeConverted.ToLower()))
         {
            return "ARR";
         }
         if (("TAKEOFF".ToLower() == toBeConverted.ToLower()))
         {
            return "DEP";
         }
         return null;
      }

      public virtual string Swap68(string toBeConverted)
      {
         if (("A10A".ToLower() == toBeConverted.ToLower()))
         {
            return "AO10A";
         }
         if (("MAF-B052H".ToLower() == toBeConverted.ToLower()))
         {
            return "CAF-B52H";
         }
         if (("MAF-F015C".ToLower() == toBeConverted.ToLower()))
         {
            return "CAF-F15C";
         }
         return null;
      }
   }
}
```