Mapping EDOC to Web Services using YATL

Octavian Patrascoiu  
Computing Laboratory, University of Kent, UK  
O.Patrascoiu@kent.ac.uk

Abstract

Modeling is a technique used extensively in industry to define software systems, the UML being the most prominent example. With the increased use of modeling techniques has come the desire to use model transformations. The current paper presents the mapping from EDOC profiles to Web Services using a transformation language called YATL (Yet Another Transformation Language). This transformation language has been defined to perform transformations within the OMG’s Model Driven Architecture (MDA) framework. After having presented YATL, we present an experiment to show how YATL can be used to map from EDOC to Web Services. YATL is still evolving since it is supposed to match the forthcoming OMG’s Query/Views/Transformations (QVT) standard.

1. Introduction

The OMG’s MDA is a new approach to develop large software systems. The core technologies of MDA are the Unified Modeling Language (UML), Meta-Object Facility (MOF), XML Meta-Data Interchange (XMI) and Common Warehouse Metamodel (CWM). These standards are used to facilitate the design, description, exchange, and storage of models. MDA also introduces other important concepts: Platform-Independent Model (PIM), Platform-Specific Model (PSM), transformation language, and transformation engine. The basic MDA pattern allows the same PIM, which specifies business system or application functionally and behavior, to be mapped automatically to one or more PSMs. While the current OMG standards such as UML and MOF provide a well-established foundation for defining PIMs and PSMs, no such well-established foundation exists for transforming PIMs to PSMs. The current paper presents YATL and the mapping from EDOC profiles to Web Services using YATL. YATL has been defined to perform transformations within the OMG’s MDA framework.

2. About KMF and YATL

The Kent Modeling Framework (KMF) [13] is being developed to provide a set of tools to support model driven software development. At the core of KMF are KMF-Studio and YATL-Studio. KMF-Studio is a tool that generates modeling tools from the definition of languages expressed as models. KMF-Studio is supported by OCL4Common and OCL4KMF, two Java libraries that allows dynamic evaluation of OCL2 constraints; and XMI, a Java implementation of the XMI standards. YATL-Studio is a tool that supports the development of transformations written in YATL, using the code generated by KMF-Studio. They use a Java library that supports reading and writing of models in XMI format.

The relations and interactions between MDA concepts in KMF are depicted in Figure 1. In our approach, the source and target models are described using the MOF language, which in this case acts like a metalanguage. The transformation language, in our case YATL, is described using two metalanguages: BNF and MOF. BNF is used to describe the concrete syntax, while MOF is used to describe the abstract syntax. The transformation engine performs the mapping from a source model instance to a target model instance, executing a YATL program, which is an instance of the YATL transformation language.

The entire transformation process is performed in KMF following the steps:

- The source and target models are defined using a MOF editor (e.g. Rational Rose or Poseidon)
- KMF-Studio is used to generate Java implementations of the source and target models.
- The source model instance is created using either Java hand-written code or the GUI provided by the modeling tool generated by KMF-Studio.
2.1 A brief description of YATL

This subsection presents the current version of YATL (Yet Another Transformation Language), which is evolving in order to support all the features provided by [14] and the future QVT standard.

YATL is a hybrid language (a mix of declarative and imperative constructions) designed to answer the Query/Views/Transformations Request For Proposals [14] issued by OMG and to express model transformations as required by the MDA [18] approach.

YATL formulates queries to interrogate the model using constructions from the OCL 2.0 standard. A YATL query is a syntactic construct that wraps inside the description of the request in terms of OCL 2.0 [20]. The YATL processor invokes the OCL processor to process the query and supply the results of interrogation.

A YATL transformation describes a mapping between a source MOF metamodel S, and a target MOF metamodel T. The transformation engine uses the mapping to generate a target model instance conforming to T from a source model instance conforming to S. The source and the target metamodels may be the same metamodel. Navigation over models is specified using OCL.

Each transformation contains one or more transformation rules. A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS of a YATL transformation is specified using a filtering expression written either in OCL or native code such as Java, C#, and scripts. This approach allows filter expressions to include both modeling information (e.g. navigational expressions, properties values, collections) and platform dependent properties (e.g. special conversion functions), which makes them extremely powerful. A compound statement specifies the effect of the RHS. The LHS and RHS for the YATL transformation are described in the same syntactical construction, called transformation rule. A rule is invoked explicitly using its name and with parameters. The body of rule $R$ is applied over every source model element for which the filter attached to rule $R$ is true. The abstract syntax of YATL namespaces, translation units, queries, views, transformations, and transformations rules is described in Figure 2.
The declarative features come mainly from OCL expressions and the description of the LHS of transformation rules. YATL acts in a similar way to a database system that uses SQL to interrogate the database and the imperative host language to process the results of the query. We choose OCL to describe the matching part of YATL rules because it is a well-defined language for querying the UML models it provides a standard library with an acceptable computational expressiveness, it is a declarative language, and it is a part of the OMG’s standards.

YATL supports several kinds of imperative features, used in the RHS of transformation rules, which are presented later in this chapter. This features were selected so that YATL can provide lifecycle operations like creation and deletion, operations to change the value of properties, declarations, decisions, and iteration statements, native statements to interact to the host machine, and build statements to ease the construction of target model instance. Compound statements contain a sequence of instructions, which are to be executed in the given order. These syntactic constructions make use of OCL expressions to specify basic operations such as adding two integer values. YATL uses the same type system as OCL 2.0 [20].

YATL is described by an abstract syntax (a MOF metamodel) and a textual concrete syntax. It does not yet have a graphical concrete syntax as QVT RFP suggested. A transformation model in YATL is expressed as a set of transformation rules. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in YATL to implement the MDA.

A YATL transformation is unidirectional. We believe that a model transformation language should be unidirectional, otherwise it cannot be used for large scale models. The main difficulty with a bidirectional transformation language is that it needs some reasoning to perform the transformation that makes the implementation slow. For example, DSTC’s proposal [15] uses mechanisms similar to Prolog-unification to perform a bidirectional mapping. The reverse transformation can be described as any other transformation using YATL.

For a real model-to-model transformation, traceability is absolutely necessary to make the approach workable. To trace the mapping between source and target model instances, YATL comprises an operator called track. Track expressions are, from the concrete syntax point of view, similar to DSTC’s track constructions [15]. The main difference is that YATL’s tracks are defined using concepts like relation name, domain, and imagine, and not Prolog-like concepts.
We experimented YATL on substantial and representative examples for clarification and validation purposes (UML class diagrams to Java classes, spider diagrams [10] to OCL, and EDOC to Business Process Execution Language (BPEL), Web Service Definition Language (WSDL), and XML Schema (XSD). In this paper we present the EDOC to Web Services mapping.

This section provides a mapping of a distributed system described using a subset of EDOC into an equivalent system described using Web Services. The subset contains only distributed systems described by EDOC’s Model Document and Component Collaboration Architecture profiles.

As models are manipulated at the abstract syntax tree level, the transformation rules were designed to obey the well-known compositional principal of Frege [11]: “the meaning of a syntactic construct is a function of the meanings of its constituents”. Each source syntactic construct is mapped to an equivalent target syntactic construct considering all its inner syntactic constructs in a bottom-up process. The source and target model instances are equivalent if they have the same black-box behavior.

The first two subsections contain a brief description of EDOC and Web Services. The subsequent sections describe the system and the transformation that performs the mapping. The entire transformation from Model Document to XML Schema is described in Appendix.

3. Transformation from a subset of EDOC to Web Services

3.1. EDOC: the UML profile for Enterprise Distributed Object Computing specification

The EDOC profile of UML was adopted by the OMG in November of 2001 as the modeling framework for Internet computing, integrating web services, messaging, ebXML, .NET and other technologies under a common technology-independent model. It comprises a set of profiles, which define the Enterprise Collaboration Architecture (ECA), the Patterns, and the Technology Specific Models and Technology Mappings.

To map from EDOC to WS we must consider the following five UML profiles:

- The Component Collaboration Architecture (CCA) uses UML classes, collaborations, and activity graphs to model the structure and behavior of components that are part of a system.
- The Entity profile describes a set of UML extensions that may be used to model entity objects.
• The Events profile describes a set of UML extensions that may be used to model event driven systems.
• The Business Process profile specializes the CCA and comprises a set of UML extensions that can be used to model business processes.
• The Relationship profile contains extensions of the UML core to rigorously specify relationships.
• The Patterns profile defines a standard means, Business Function Object Patterns that can be used to describe object models using the UML package notation.
• The Technology Specific Models and the Technology Specific Mappings take into account the mapping from ECA specification to technology specific models. It defines and EDOC profile for Enterprise Java Beans (EJB) and another for Flow Composition Model (FCM).

3.2 Web Service

The purpose of web services is to enable a distributed environment in which any number of applications, or application components, can communicate in a platform-independent, language-independent fashion. A web service is a piece of software application, located on the Internet that is accessible through standard-based Internet protocols such as HTTP or SMTP.

Given this definition, several technologies used in recent years could have been classified as web service technologies, but were not. These technologies include win32 technologies, J2EE, CORBA, and CGI scripting. These technologies are not web services technologies mainly because they are based on a proprietary binary standard, which is not supported globally by most major technologies firms. The core of the web services technologies is made of eXtensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL), and Universal Description, Discovery and Integration (UDDI).

XML is a widely used standard from the World Wide Web Consortium (W3C) that facilitates the interchange of data between computer applications. XML uses markup codes (tags) to describe data, just like the HyperText Markup Language (HTML) that is used to describe Web pages. Computer programs can automatically extract data from an XML document, using its associated DTD as a guide.

SOAP provides a standard packaging structure for exchanging XML documents over a variety of Internet protocols, including HTTP, SMTP, and FTP. The existence of a standard transport mechanism allows heterogeneous clients and servers to communicate. For example, .NET clients can invoke EJBs and Java clients can invoke .NET Components through SOAP.

WSDL is an XML technology that provides a standard description of web services. WSDL can be used to describe the representation of input and output parameters of an invocation, the function’s structure, the nature of the invocation, and the protocol used for transport.

UDDI provides a worldwide registry of web services for description, discovery, and integration purposes. Analysts and technologist use UDDI to discover available web services by searching for categories, names or identifiers.

3.3 Mapping from Document Model to XML Schema

Both EDOC and WS models describe business processes. A business process manipulates and exchange information with other business processes. To describe the information that is manipulated or exchanged during a business process, both EDOC and WS have dedicated components: Model Document and XML Schema respectively.

The first step in the mapping from EDOC to WS is to map the models that are used to describe the information that is manipulated. This section contains the description of the mapping process from Model Document to XML Schema.

The Document Model package from the EDOC profile defines the information that can be manipulated by EDOC ProcessComponents. The document model is based in data elements that can be either primitive data types or composite data. A CompositeData contains several attributes. An attribute has a specific type, an initial value and can be marked as required or as many to indicate the cardinality. An enumeration defines a type with a fixed set of values. The document model is described in Figure 3. The XML Schema [27] describes the information that can be manipulated by web services. It contains types that can be simple, such as string or decimal, or complex. A ComplexType contains a sequence of attributes. An Attribute has a name and a given type. A partial model of XML Schema is given in Figure 4.

It is obvious that mapping from Model Document to XML Schema means mapping from Element, DataType and CompositeData to Type, SimpleType
and ComplexType respectively. The transformation process and the rules that perform the mapping are described briefly in Table 1.

**Table 1: Mapping Document Model to XML Schema**

<table>
<thead>
<tr>
<th>Rule name</th>
<th>Rule description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt2st</td>
<td>Creates a XML Schema SimpleType for each Document Model DataType and stores the mapping using the track mechanism.</td>
</tr>
<tr>
<td>cd2ct</td>
<td>Creates a XML Schema ComplexType for each Document Model CompositeData and stores the mapping using the track mechanism.</td>
</tr>
<tr>
<td>at2at</td>
<td>Creates a XML Schema Attribute for each Document Model Attribute and stores the mapping using the track mechanism.</td>
</tr>
</tbody>
</table>
3.4 Mapping from CCA to WSDL

The CCA profile details how the UML concepts of classes and collaboration graphs can be used to model the structure and the behavior of the components that comprise a system. In CCA process components interact with other process components using a set of ports. A ProcessComponent describes the contract for a component that performs actions. A Port defines a point of interaction between process components. Ports can be classified according to the complexity of the interaction in FlowPorts, ProtocolPorts, OperationPorts, and MultiPorts. A FlowPort is a port capable to produce and consume a single data type. ProtocolPorts describe more complex interactions based on Protocols. A Protocol is a method by which two components can communicate. An OperationPort is a port that realizes a typical request/response operation. A MultiPort is a group of ports whose actions are tied together. The specification of a ProcessComponent may include a Choreography to specify the sequence of interactions performed through ports. In WSDL the Definition element acts as a container for the service description. The Import element serves a purpose similar to the "#include" directive in the C/C++ programming language. It lets the modeler separate the elements of a service definition into separate documents and include them in the main document. The Type element acts as a container for the definition of datatypes that are used in the Message elements. The Message element is used to model the data exchanged in a web service. A message is made of several parts, each part having a name and a type. The PortType element specifies a subset of operations supported for an endpoint of a web service. The Operation element models an operation. A WSDL operation can have input, output, and fault messages as part of its action. The Binding element specifies the protocol and data format of a PortType element. The bindings can be standard - HTTP, SOAP, or MIME – or can be created by the user. The Service element typically appears at the end of a WSDL document and identifies a web service. The primary purpose of a WSDL document is to describe the abstract interface. A Service element is used only to describe the actual endpoint of a service. Figure 5 contains the WSDL model.

The transformation process and transformation rules are described in Table 2.

Table 2 Transformation from CCA to WDSL

<table>
<thead>
<tr>
<th>Rule name</th>
<th>Rule description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkAttribute2Type</td>
<td>Sets the correct value for type property for each XML Schema Attribute.</td>
</tr>
<tr>
<td>linkComplexType2Attribute</td>
<td>Sets the correct value for sequence property for each XML Schema CompositeType</td>
</tr>
<tr>
<td>documentModel2xsd</td>
<td>Invokes the above rules in the following order: apply dt2st(); apply cd2ct();</td>
</tr>
<tr>
<td></td>
<td>apply at2at(); apply linkAttribute2Type(); apply linkComplexType2Attribute();</td>
</tr>
</tbody>
</table>

flowPort2message

Creates a WSDL Message for each CCA FlowPort and stores the mapping using the track mechanism.

operationPort2operation

Creates a WSDL Operation for each CCA OperationPort and stores the mapping using the track mechanism. The input and output properties of the WSDL Operation are computed using the initiator and the responder port from the OperationPort.

protocolPort2portType

Creates a WSDL PortType for each CCA ProtocolPort and stores the mapping using the track mechanism.

processComponent2service

Creates a WSDL Service for each CCA ProcessComponent and stores the mapping using the track mechanism. The definition of the service is instantiated by this rule. The values of the properties are assigned by the other rules.
3.5 An example

To study and test the mapping from EDOC to WS using YATL and YATL-Studio we consider the transformation of the EDOC model of a travel agency into the equivalent description that uses Web Services concepts. In general a travel agency provides services such as: reserves and purchases flights and charters tickets, reserves hotel rooms, rents cars, books holidays and cruises, and sells travel insurance. To provide such services a travel agency needs to establish business links with companies such as airlines, hotels, and banks. Due to lack of space we present only a brief description of the system.

Figure 6 contains the description of a travel agency community process. The activities in the TravelAgency Community Process start by the Client initiating the interactions on its Buy ProtocolPort, according to the BuySell protocol. The TravelAgency is connected through the Sell ProtocolPort with the Client and responds to the BuySell protocol initiated by the Client. The TravelAgency uses the dedicated ports BuyFlight, ReserveRoom, RentCar, and Payment to communicate with the other processes: Airline, Hotel, CarCompany, and Bank. The TravelAgency initiates the communication through these ports, according to Client’s requests.
Figure 6 Travel agency community process

Figure 7 BuySell and BuyFlight choreography

Figure 7 contains the description of choreographies for BuySell and BuyFlight protocols. Similar choreographies can be derived for ReserveRoom.

The Appendix contains, due to lack of space, a partial description of the transformation rules that perform the mapping from EDOC to WS. A detailed description of the transformation rules is presented in [24].

The transformation was performed in KMF using KMF-Studio and YATL-Studio. First KMF-Studio was used to generate Java code corresponding to model elements both for source and target model (EDOC and WS). This code and a textual description of the transformation rules were used by YATL-Studio to create a target model instance from a source model.
instance. All the transformations were performed at the syntax tree level.

4. Conclusions and future work

This paper has shown a technique for model transformations based on a transformation language called YATL. Model transformation can be described using a variety of transformation techniques [1][4][11][25]. The PROgrammed GRaph REplacement System (PROGRES) [25] contains means not only to specify transformation rules but also to define the sequencing of these rules (described using imperative constructs). This features of PROGRES sets it apart from many of the other graph transformation approaches. Unfortunately, PROGRES provides no direct support for UML. Another graph transformation system for domain-specific model transformations is the Graph Rewriting and Transformation Language (GReAT for short) [11]. Similarly to PROGRES, it separates the language for describing transformation rules from the language for describing rule ordering. Unfortunately the execution engine of GReAT is slow, which makes the language unusable in industrial environment. AGG [1] does not provide sufficiently rich mechanisms for controlling the application of transformation rules.

Comparing to these languages YATL is simple, easy to learn and understand, uses OO and UML concepts, and has a high descriptive power. To test YATL’s descriptive power and its expressiveness we performed several transformations. These experiments [24], especially the EDOC to WS mapping, forced us to add new features to YATL and improve the implementation. They also proved that YATL can be used to described complex transformations for large scale systems, it is easy to use, easy to learn and understand as it is described using OO concepts and a mix of procedural and non-procedural constructs.

The transformation that we presented in this paper maps only a subset of EDOC to WS. The intention is to provide a complete mapping from EDOC to WS. In the near future we intend to study the mapping of the dynamic part (choreography) of EDOC models to web services, to compare the two approaches in terms of their description power and expressiveness, and to study the limits of this transformation.

YATL is still evolving because one of our main goals is to make it complaint to the QVT standard. But we also hope to add many original features to the YATL development environment, to integrate it with KMF-Studio and provide support for transformations using YATL in IBM’s Eclipse Modeling Framework (EMF).

5. References

[15] MOF Query/Views/Transformation, Initial submission, DSTC and IBM.

Appendix

```c
start kmf::edoc2ws::main;
namespace kmf(sd, ocl) {
    transformation edoc2ws {
        -- EDOC.ECA.DocumentModel to WS.XSD
        rule dt2st match edoc::ECA::DocumentModel::DataType () {
            -- Create SimpleType and store mapping
            let st: ws::xsd::SimpleType;
            st := new ws::xsd::SimpleType;
            st.name := self.name;
            track(self, type2type, st);
        }
        -- Map an EDOC CompositeData to an XSD ComplexType
        rule cd2ct match edoc::ECA::DocumentModel::CompositeData () {
            -- Create ComplexType and store mapping
            let ct: ws::xsd::ComplexType;
            ct := new ws::xsd::ComplexType;
            ct.name := self.name;
            track(self, type2type, ct);
        }
        -- Map an EDOC Attribute to an XSD attribute
        rule at2at match edoc::ECA::DocumentModel::Attribute () {
            -- Create Attribute and store mapping
            let at: ws::xsd::Attribute;
            at := new ws::xsd::Attribute;
            at.name := self.name;
            track(self, at2at, at);
        }
        -- Link XSD attributes to XSD types
        rule linkAttribute2Type match edoc::ECA::DocumentModel::Attribute () {
            -- Get the XSD Attribute
            let xsdAttribute: ws::xsd::Attribute;
            xsdAttribute := track(self, at2at, null);
            -- Get the type
            let edocType: edoc::ECA::DocumentModel::DataElement;
            edocType := self.type;
            let xsdType: ws::xsd::Type;
            xsdType := track(edocType, type2type, null);
            xsdAttribute.type := xsdType;
        }
        -- Link XSD ComplexTypes to XSD Attributes
        rule linkComplexType2Attribute match edoc::ECA::DocumentModel::CompositeData () {
            -- Get the XSD ComplexType
            let xsdComplexType: ws::xsd::ComplexType;
            xsdComplexType := track(self, type2type, null);
            -- Add every attribute
            foreach edocAttribute: edoc::ECA::DocumentModel::Attribute in self.features do {
                let xsdAttribute : ws::xsd::Attribute;
                xsdAttribute := track(edocAttribute, at2at, null);
                xsdComplexType.sequence := xsdComplexType.sequence->including(xsdAttribute);
            }
    }
}
```
-- Map concepts from EDOC.ECA.DocumentModel to WS.XSD concepts
rule documentModel2xsd() {
    -- Create a SimpleType for each DataType
    apply dt2st();
    -- Create a ComplexType for each CompositeData
    apply cd2ct();
    -- Create an XSD Attribute for each EDOC Attribute
    apply at2at();
    -- Link XSD Attributes to XSD Types
    apply linkAttribute2Type();
    -- Link XSD ComplexTypes to XSD Attributes
    apply linkComplexType2Attribute();
}

-- Map concepts from EDOC.ECA.CCA to WS:WSDL
rule flowPort2message match edoc::ECA::CCA::FlowPort () {
    -- Create Message
    -- Create part and add it
    -- Store mapping
}

rule operationPort2operation match edoc::ECA::CCA::OperationPort () {
    -- Get input and output port
    -- Create input
    -- Create output
    -- Create Operation
    -- Store mapping
}

rule protocolPort2portType match edoc::ECA::CCA::ProtocolPort () {
    -- Create a portType
    -- Add operations
    -- Store mapping
}

rule processComponent2service match edoc::ECA::CCA::ProcessComponent () {
    -- Create Definition
    -- Create service
    -- Store mapping
}

rule linkDefinition2X match edoc::ECA::CCA::ProcessComponent () {
    -- Get the WSDL Service
    -- Add every portType
}

--- Map CCA to WSDL
rule cca2wsdl() {
    -- Create a WSDL Message for each EDOC FlowPort
    apply flowPort2message();
    -- Map Operation Ports
    apply operationPort2operation();
    -- Map Protocol Ports
    apply protocolPort2portType();
    -- Map ProcessComponent
    apply processComponent2service();
    -- Link Definition to types, messages, and portTypes
    apply linkDefinition2X();
}

-- main rule
rule main () {
    -- Map DocumentModel to XSD
    apply documentModel2xsd();
    -- ECA to WSDL
    apply cca2wsdl();
}