

Duddy, Keith R., Henderson, Michael S., Metke-Jiminez, Alejandro, & Steel, Jim (2010) ***Design of a model-generated repository as a service for USDL***. In: Proceedings of the 1st International Workshop on the Internet of Services, 8-10 November 2010, IUFM University Cergy-Pontoise, Paris.

# Design of a Model-Generated Repository as a Service for USDL *

Keith Duddy
Queensland University of Technology
2 George St
Brisbane
QLD 4000
Australia
keith.duddy@qut.edu.au

Michael Henderson
Queensland University of Technology
2 George St
Brisbane
QLD 4000
Australia
michael.henderson@qut.edu.au

Alejandro Metke-Jimenez
Queensland University of Technology
2 George St
Brisbane
QLD 4000
Australia
a.metke@qut.edu.au

Jim Steel
Queensland University of Technology
2 George St
Brisbane
QLD 4000
Australia
james.steel@qut.edu.au

## ABSTRACT

SAP and its research partners have been developing a language for describing details of Services from various viewpoints called the Unified Service Description Language (USDL) [12]. At the time of writing, version 3.0 describes technical implementation aspects of services, as well as stakeholders, pricing, lifecycle, and availability. Work is also underway to address other business and legal aspects of services. This language is designed to be used in service portfolio management, with a repository of service descriptions being available to various stakeholders in an organisation to allow for service prioritisation, development, deployment and lifecycle management.

The structure of the USDL metadata is specified using an object-oriented metamodel that conforms to UML, MOF and EMF Ecore. As such it is amenable to code generation for implementations of repositories that store service description instances. Although Web services toolkits can be used to make these programming language objects available as a set of Web services, the practicalities of writing distributed clients against over one hundred class definitions, containing several hundred attributes, will make for very large WSDL interfaces and highly inefficient "chatty" implementations.

This paper gives the high-level design for a completely model-generated repository for any version of USDL (or any other data-only metamodel), which uses the Eclipse Modelling Framework's Java code generation, along with several open source plugins to create a robust, transactional repository running in a Java application with a relational datastore. However, the repository exposes a generated WSDL interface at a coarse granularity, suitable for distributed client code and user-interface creation. It uses heuristics to drive code generation to bridge between the Web service and EMF granularities.

## Keywords

Services, Web Services, USDL, Service Description, Model Driven Engineering

## 1. INTRODUCTION

The Unified Service Description Language (USDL) is a specification in progress which attempts to capture metadata in a standard form about *services*, in their broadest possible sense. USDL covers the space of business, technical, stakeholder, legal and lifecycle aspects of services, as well as sets of relationships among the services described. It began as an SAP project that specified a number of XML schemata for the capture of some of the business aspects of services, over and above the usual technical interface information usually stored in middleware repository technologies such as UDDI [9] or the IBM WebSphere Service Registry and Repository. USDL has since seen contributions from partner organisations of SAP, including the authors of this paper, and has evolved into an object-oriented metamodel. At the time of writing, SAP is setting up an industry consortium known as *Internet of Services*[1] to foster the develop-

*The work reported in this paper has been funded in part by the Smart Services Co-operative Research Centre through the Australian Federal Government's CRC Programme (Department of Innovation, Industry, Science and Research).



[1]Web site: http://internet-of-services.com

ment and uptake of the USDL specification, among others, at arm's length from the company. The Internet of Services organisation has the goal of soliciting wide services industry input and support, and fostering eventual international standardisation with an appropriate standards body.

Considering the rapid evolution of the USDL specification, the authors consider that the only viable approach to the creation of trial repositories for testing intermediate versions of USDL is to use model-driven approaches, as hand modification of implementations is expensive and error prone. However, existing tools for code generation from UML [2] or EMF [3] *ecore* models apply a naive approach which generates a programming language class for every metamodel class, resulting in a very fine grained API to manipulate the service descriptions. This is a viable approach when coding a non-distributed application with appropriate user interfaces. However, these types of APIs are unsuited to distributed programming, and in the kinds of organisations that manage large portfolios of services, the requirement for distributed access to such repositories via custom clients using Web services is a given.

The USDL specification in version 3.0 has been published via the Internet of Services web site. The site will become the home of the industry consortium being established. It contains six model packages, and preempts the publication of two yet to be completed packages: "legal" and "service level". Figure 1 shows the proposed structure for a complete USDL specification.

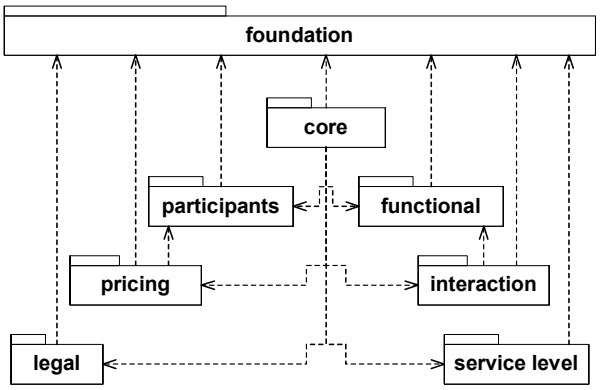


**Figure 1: USDL Module Structure**

## 1.1 Structure of this Paper

The goals and structure of USDL, and the motivation for a Web services based repository for storing service descriptions is given in this Introduction section. The paper then provides a set of primary use cases for a USDL repository, with three high level user roles defined in section 2.

Section 3 is the main part of the paper which provides a detailed rationale for a coarse-grained Web services based repository. It then provides a high-level architecture for the construction of such a repository. It introduces a number of model-driven technologies that are used to automatically generate parts of the repository, and outlines the additional model transformation and code generation techniques required to provide the functionality needed.

Finally, section 4 provides a brief overview of additional USDL model management, and repository synchronisation work being conducted in the same project.

# 2. PRIMARY USE CASES

Figure 2 shows five use cases, related to three user roles for a USDL Repository. These are given a brief overview in the following subsections.

## 2.1 Roles

The following roles (Actors in UML parlance) are envisaged for a USDL repository deployed in an organisation with shared services. Additional roles will be needed to describe the interactions with a repository deployed in a networked organisation or marketplace.

**Service Owner** A role played by those with responsibility for the deployment and operation of a service, and hence with its primary description in the repository.

**Service Portfolio Manager** A role played by strategic management of services planning, provisioning coordination and lifecycle. Usually played by Enterprise Architects or those serving CIOs.

**Service User** Any person requiring the use of a service or services in the creation of applications or processes.

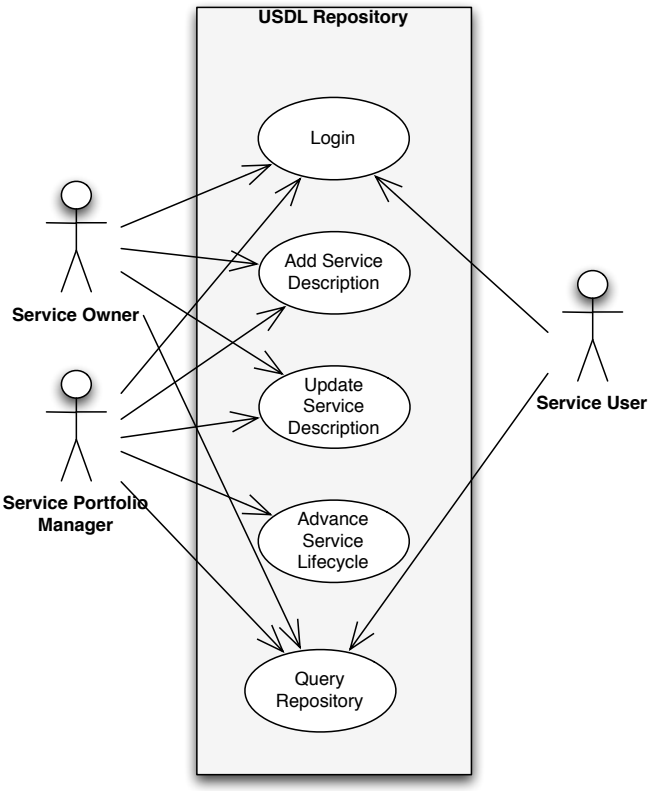


**Figure 2: USDL Repository Use Cases**

## 2.2 Login

This use case implies that access to the repository is controlled, and authentication of users is required for access to it. Each user may be assigned a role, and initially we use the three Actors in the use cases as the set of available roles.

## 2.3 Add Service Description

This use case is available to actors playing the Service Owner and the Service Portfolio Manager roles. Its name is self-explanatory.

## 2.4 Update Service Description

This use case is available to actors playing the Service Owner and the Service Portfolio Manager roles. Updating a service description may include changing various attributes describing the service, or adding more implementation detail as a service progresses from planning to detailed design to implementation and deployment.

## 2.5 Advance Service Lifecycle

Attributes of a USDL service description allow for various lifecycle stages to be explicitly represented, and these can be progressed only by those in the Service Portfolio Manager role.

## 2.6 Query Repository

This use case is accessible to all actors, but the set of services exposed through the interface will be different for each role. Service Portfolio Managers will be able to query all service descriptions, while Service Owners will be able to see all of the services they own, in any lifecycle state. Whereas Service Users will be able to see only services that are deployed for use, and not yet deprecated.

# 3. BUILDING REPOSITORIES USING A MODEL DRIVEN APPROACH

In order to provide the most agile approach available for the implementation of USDL repositories, a model of USDL service descriptions has been created in the Eclipse Modelling Framework (EMF). EMF comes with various capabilities for generating code to support repositories for instances of the USDL model. Specifically, it supports the generation of Java code that provides object implementations for in-memory representations of model instances, as well as the ability the serialise these instances as XML in plain-text files. It also generates default tree-based editors for model instances.

However, the repository that we need to generate is a three-tiered distributed application with a persistent storage layer, an in-memory object model, and a distribution layer. See Figure 3 for a digramatic representation of a complete Repository as a Service, as well as the technologies that generate various components from the Ecore model.

EMF provides us with the machinery to generate the in-memory Java objects for the middle tier. The other tiers require additional technologies for generation of their implementations.

## 3.1 Generation of Persistent Storage

Additional Java/Eclipse plugins allow us to use model-driven generation to achieve the storage layer:

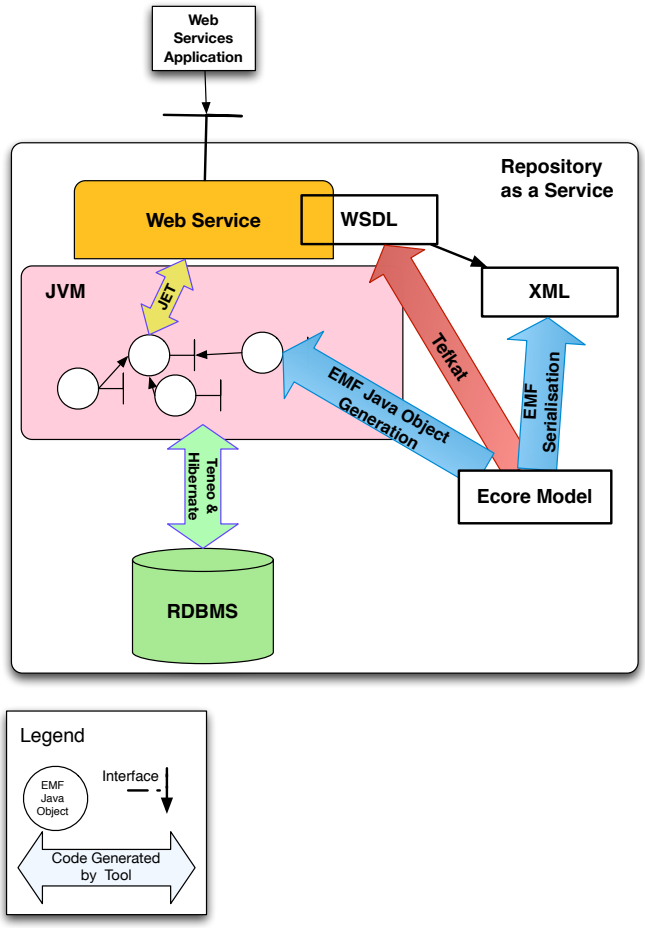


**Figure 3: USDL Repository Architecture**

**Hibernate** Allows Java classes to be persisted to relational databases by making annotations.

**Teneo [11]** Uses the EMF framework to auto-generate the Hibernate annotations for generated classes to map the object model to relational tables.

With an appropriate build environment, and some annotations in place, the EMF model of USDL can be used to fully generate in-memory and persistent storage, in both XML and relational forms, for instances of USDL service descriptions, using EMF and Teneo in combination.

## 3.2 Generation of an API

The USDL model contains too many classes and references to use the EMF generated Java Object APIs as a basis for interacting with the repository by invoking getters, setters, and reference navigation on each class. A coarser-grained interface is necessary to deploy the API as a service available to distributed programmers. The argument against very fine grained CRUD APIs for distributed applications is well understood in the software development community [7], and strategies for coping with this problem are exemplified by design patterns such as Sun's EJB pattern "Composite Entity" [5].

Our approach to designing a Web services API for the USDL model at a coarser granularity than the Ecore model involves a grouping of classes and their attributes and references into larger units that can each be represented by a WSDL interface type. In addition, some parts of the model are about relationships between the data entities (for example Service Dependency graphs), and these will be grouped and managed by the creation of additional WSDL interfaces.

We wish to generate this larger granularity API automatically using declarative model to model transformations. However, the semantics of a domain-specific model, such as USDL, means that some domain-specialist-specified direction of the transformations will yield better results in some cases. In addition, our implementation approach will be suitable for generating Web service APIs to any data-centric metamodel.

The approach to grouping classes in the input model for the purposes of generating WSDL interfaces is as follows:

1. All abstract and attribute-less base classes in the model are ignored for interface generation, as these are never instantiated. Although, of course, the attributes and associations defined by them are treated as a part of any of their concrete derived classes.

2. The "primary root" class of the model in question must be nominated by the domain expert. In the case of the USDL model in its current form, this is the "Service" class, which has attributes of atomic data types and references to other classes, generally in the form of compositions of classes (using composite, or black-diamond, references) that compose additional classes in a tree structure. For example, a service has zero or more PricePlans, which have one or more PriceComponents, which have one or more Price Levels, and so on. The generated WSDL will then contain a primary interface for this root class, and an operation to allow the creation of all of its attributes and composed classes (and their attributes and composed classes, recursively) in a single invocation. The input parameter type for this operation is generated by EMF XML serialisation, built in to the platform. The output parameter is a unique identifier for the root object created by a successful invocation, and several common faults may be raised.

   Similarly, a delete operation is created, which takes the unique identifier as its input parameter. At the moment, the update operation is a double of the create operation with an additional identifier parameter, but future plans include the option of generating a thin WSDL wrapper for the fine-grained EMF generated classes to allow single attributes to be updated without the transmission of all the other values which remain unchanged.

3. We cannot, however, represent all properties of a service as trees of objects containing attributes, aggregated by other objects, as this implies that each property of the service is unique to that service description, and its lifecycle is bound to the lifecycle of the service description. An example is the representation of the stakeholders of the service: represented by the abstract "Agent" class and its subtypes "NaturalPerson" and "Organization". We would not wish to represent a service provider, or owner as a separate object for each service description, as we need to be able to update the attributes of these objects in one place only. This means we can change, for example, an address or phone number, once, and have each service description that refers to the object share this change.

   Therefore, we must also identify the other "root" classes of the model, representing shared metadata, so that we can allow the generated API to create and update instances of these classes independently. Any complex metadata model is effectively a forest with non-composition relationships between nodes in the trees. Our current approach to WSDL generation creates a separate interface for each root class that has an aggregation tree of more than two concrete classes, and a general factory interface for the creation of simple shared objects. The interface generation proceeds as described above for the primary root object.

4. The non-composition references in the model are given their own WSDL interfaces on a package by package basis. An operation is created per reference simply taking two parameters representing the unique identifiers of the source and target (or set of targets in the case of multi-valued references). The operations are located in the interface for the package of the target end of the reference in the case of cross-package references, as this provides a better logical grouping; references to price-related classes added in the price interface, references to legal-related classes in the legal interface, and so on.

5. There are a number of cases in the USDL model where non-composite references to classes are used (correctly or incorrectly) to representing concepts that are not really shared or truely re-usable when instantiated as objects. A good example is the "TechnicalCondition" class, which is used in the model to provide expressions constraining technical interfaces. It is unlikely that the use of an expression will be stored in the repository

only once and shared in all uses; it is more likely that each expression will be written for the specific interface, even if it is identical to one already stored in the repository. Therefore, for these kinds of cases, the user of the repository generator may mark these references as "effectively composite", and treat them in the same way as composition references for the purposes of operation and parameter generation, as described in (2) above. In this case, the factory operations for the class will not be generated, and the object's lifecycles will correspond to those of their de-facto parent objects.

## 3.3 Code Generation for the Distribution Layer

The code generation for the WSDL and its implementation is achieved through several tools. We re-use existing plugins where possible, and use model to model or model to text transformation engines where we need new approaches.

The WSDL API generation heuristics are implemented through a model transformation, expressed in the Tefkat [8] model transformation language for EMF, on which the authors have worked previously. Tefkat is available through an open source project[2]. The transformation would be responsible only for the relatively simple task of creating appropriately named interfaces and operations in a WSDL Ecore metamodel [3] which transmit and receive these types. This deals with the ability to add, modify and delete parts of the description of a service in USDL. The WSDL Ecore model has an eclipse resource implementation that allows the WSDL text files to be output automatically. However, code generation for languages with EMF models and resource implementations may also be achieved using model to model transformations. One such example is OCL, for which such an implementation exists.

We also use EMF itself to generate and export XML schemas that correspond to the structure of a set of classes and their attributes and relationships. EMF generated code will also be used to de-serialise the XML arguments received by Web service invocations, and re-serialise any returned results. We have developed an Eclipse plugin that facilitates this process with the provision of wizards to obtain relevant configuration parameters from users.

Finally we use JET to generate the java code that matches up the object values corresponding to the XML parameters received at the Web service with the appropriate EMF operations to create, update and delete the fine grained EMF objects and set their attribute values and references. The JET code and templates interrogate the extensive traceability models that are kept by Tefkat as a record of the correspondences between the input USDL model and the generated WSDL model.

## 3.4 Query

The next problem which requires a solution is the ability to query a set of services in the USDL repository to match appropriate service descriptions based on some criteria. Presently we use the Hibernate Query Language (HQL) to make queries using encapsulated strings as a parameters to WSDL operations. This is only a temporary solution, as it requires the user to be aware of the Java classes that have been generated by EMF and annotated for Hibernate by Teneo.

[2]http://tefkat.sourceforge.net

[3]available at http://www.eclipse.org/webtools/ws/

It is not within the scope of this project to invent a new object query language. Therefore this task comes down to choosing an appropriate existing mechanism for finding matching services. Some possible query languages include:

**XPATH [4]** An expression language for nominating an XML element in a document structure, which could be embedded in a simple expression language to make predicates which specify matches to service descriptions.

**OCL [13]** The Object Constraint Language is targeted at the UML Class language, but as EMF is a subset of this language, could be used to write queries matching USDL service descriptions. There are existing OCL implementations for EMF.

**SPARQL [10]** An SQL-like query language for RDF which might be able to be applied to EMF.

# 4. RELATED AND FUTURE WORK

To our knowledge there is no comparable work which automatically derives medium-grained distributed programming interfaces from fine-grained object models. Current approaches allow for automated serialisation of a single programming language object, or for the parsing and processing of XML documents that are designed for transmission over a network. This results in either a lot of manual "glue code" being written, or in an inefficient use of "chatty" distributed invocation protocols.

The remainder of this section discusses complementary work being undertaken or planned in our project.

## 4.1 Model Synchronisation

Our initial case studies for service description repositories in the banking and government sectors revealed a long history of attempts to maintain service catalogues and repositories by enterprise architects and middleware development groups. To date, all of the initiatives in our partner organisations have failed, and one overwhelming reason was identified for this: after the initial population of a repository or catalogue, the data was allowed to go stale. Users of the catalogue found that over time the results they retrieved from it were becoming more and more out of date, and they stopped using the catalogue. Eventually the catalogue became disused as users returned to their old mechanisms of requesting information from colleagues by phone or email.

Based on this experience, the authors are currently investigating the use of model synchronisation components, known as Live Model Pointers [6]. These components will act as additional clients to the generated Repository, but ones which have been configured to source the data to populate the service descriptions from other master data sources which are maintained as a part of the normal operating procedures of the organisation. For example:

**LDAP and CRM** The information in USDL about people and organisations is routinely kept up to date in local and remote LDAP repositories in the case of units and people inside the organisation, and in Customer Relationship Management software for external people and organisations.

**WSDL and UML Repositories** Up to date information about the design and concrete interface specifications

for a service are always up to date in implementation-based repositories used in development teams, and synchronisation with these upon release triggers will ensure that USDL descriptions are always up to date. Whereas making a requirement that development teams update the USDL repository in addition to their design and code repositories is likely to fail.

**RDBMS, XML files, etc** Many different databases can be queried to retrieve master data.

Live Model Pointers would contain the following elements:

- A nominated set of attributes in the USDL model that form the "foreign key" of the equivalent concept in the remote repository
- A URI or other invocation endpoint on which to connect to the remote repository
- A Mapping from elements in the returned result to the attributes in the USDL model
- A number of policy settings: caching, timeouts, etc

Once again, by implementing a few generic adapters, the information above would be captured by a configuration model, and the synchronisation code would be generated.

## 4.2 Model Evolution and Domain Subsetting Methodology

When we applied the Generic USDL 2.0 (GUSDL) to the financial domain, during a case study at a bank, an appropriate subset of the whole USDL model was created to meet the needs of the client, which we call Financial USDL (FUSDL). In Figure 4 this is shown by the top-most arrow. Then additional model elements for the domain were added to the model (shown in Figure 4 in blue), and the implementation was altered to support this new Domain-specific USDL (DUSDL) model variant. This required substantial human effort, as the code for the database, and for the user interface were updated by hand.

In general, we envisage that a similar process will be required almost every time a USDL repository will be deployed in a specific domain, X, (shown as XUSDL in the figure). An additional step in the development process is required for the next DUSDL to be created for another domain, namely that the useful generic concepts added in to the financial domain version of USDL need to be re-integrated into the Generic USDL model (shown as the diagonal arrow producing GUSDL 2.1 in the figure). Then the process can begin again. We will follow this approach again, manually, for another specific domain, and then attempt to generalise the approach using model transformation techniques.

There are tools available in the EMF open source community, such as EMF Compare [1] that allow for automated model differencing and merging. It is envisaged that these tools may be built upon to allow for a graphical interface that supports the methodology. The extended tool will have two functions, of which the first is currently implemented in a prototype.

1. In the creation of a new Domain USDL, unneeded classes in the USDL model are annotated, and then the tool eliminates them in a manner that maintains model integrity by patching the inheritance hierarchy,

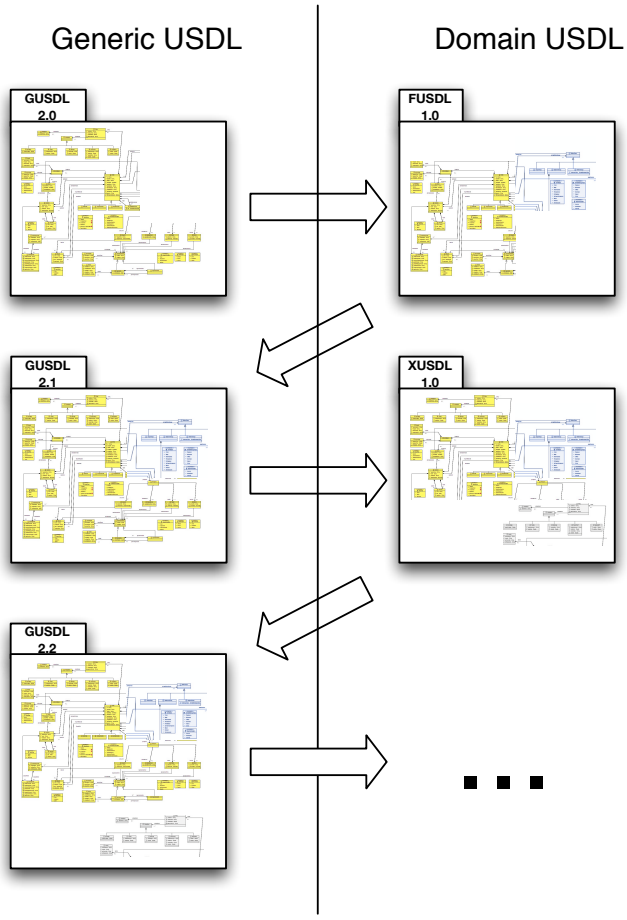


**Figure 4: USDL Methodology - High Level View**

and removing references in other classes to the removed classes. We envisage creating a graphical model editor version of this tool for ease of use.

2. When that DUSDL is augmented with new domain concept classes, the tool will be able to show the added elements, and allow the GUSDL model custodians to select the ones which should be merged back into the GUSDL.

## 5. CONCLUSION

When creating repositories for service descriptions that are specified by an object-oriented metamodel, we have found that generating a Java implementation from the model of the metadata is possible. Furthermore, much of the tooling required to construct the objects and their corresponding relational storage tables is available off the shelf. However, the environments into which we will deploy these repositories require distributed access via Web services, and this means that a different granularity of access is needed to facilitate efficient distributed invocations. To this end, we have designed a method for the creation of coarse-grained WSDL interfaces from fine-grained Ecore object models, and using a combination of code generation tools and model transformation specifications, we can automatically generate the code to bind the Web service to the Java repository.

## 6. REFERENCES


[1] M. Alanen and I. Porres. Presentation of EMF Compare utility. In *Eclipse Modelling Symposium at ESE 2006*, 2006.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[3] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[4] J. Clark and S. DeRose. XML path language (XPath) version 1.0 w3c recommendation. Technical report, World Wide Web Consortium, 1999.

[5] Core J2EE Patterns - Composite Entity. http://java.sun.com/blueprints/corej2eepatterns/Patterns/CompositeEntity.html, 2001-2002.

[6] K. Duddy. Live model pointers - a requirement for future model repositories. In *KISS Workshop at ASWEC 2009*. Industrialized Software, 2009.

[7] D. Helton. Coarse-grained components as an alternative to component frameworks. In *Proceedings of the Workshop on Object-Oriented Technology*, page 188, London, UK, 1999. Springer-Verlag.

[8] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In *Satellite Events at the MoDELS 2005 Conference*, volume LNCS Vol. 3844, 2005.

[9] E. Newcomer. *Understanding Web Services - XML, WSDL, SOAP, and UDDI*. Independent Technology Guides, 2002.

[10] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.

[11] Teneo. http://wiki.eclipse.org/Teneo, 2010.

[12] USDL Information Sheet. http://internet-of-services.com/uploads/media/USDL-Information-Sheet.pdf, 2009.

[13] J. Warmer and A. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley, 2003.