Model-Driven Reengineering Legacy Software Systems to Web Services

ABSTRACT

The advancement of internet technology enables legacy software systems to be reused across geographical boundaries. Web Services (WS) have emerged as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA), the open standard description language XML and transportation protocol HTML. Therefore, legacy software systems can incorporate WS technology in order to be reused and integrated in a distributed environment across heterogeneous platforms. In this paper, we present a comprehensive, systematic, automatable approach toward reengineering legacy software systems to WS applications, rather than rewriting the whole legacy software system from scratch in an ad-hoc manner.

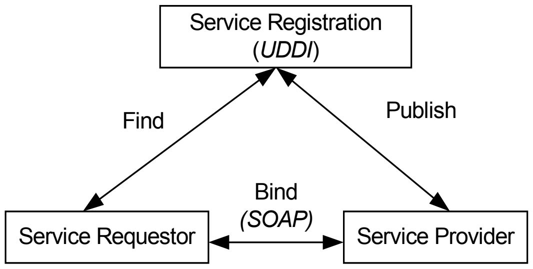
Keywords: software system reengineering; Web Services; Model-Integrated Computing; meta-model; model; model marshaling and unmarshaling; Entity-Relationship model

INTRODUCTION

Web Services as a Presentation Layer for Legacy Software Reuse and Integration

With the rapid advancement of software technology, more and more software systems developed with the state-of-the-art technologies of yesterday are becoming legacy software systems of today. Specifically, we define legacy software in a comparative manner, i.e., the software systems are *legacy* if the languages, models or platforms they are developed with can be replaced with new languages, models or platforms of advanced features and improved capabilities. The reuse and integration of legacy software systems offer a promising direction for boosting productivity by dramatically reducing both cost and time-to-market expenses (Devanbu et al., 1996). With the emergence and advancement of Internet technology, the power of legacy software systems is being unleashed toward a broader scope. Particularly, Web Services (WS) have emerged as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA) (Colan, 2004) as is illustrated in Figure 1. By using standard XML as the description language and HTTP as the transport protocol, WS can be used to wrap legacy software systems for integration beyond the enterprise boundary across heterogeneous platforms. To be specific, WS uses the XML based XML-based Web Services Description Language (WSDL) for specifying services, SOAP (Simple Object Access Protocol) messages for service invocation, and UDDI (Universal Description, Discovery and Integration) registry for service discovery (Colan, 2004). With the wrapping by WS, the integration of legacy software systems is simplified, from one to one interoperation to interoperate on the one common ground (WS).

Figure 1. Service Oriented Architecture (SOA)



Approaches for Using Web Services as a Wrapper

There are several options for reengineering legacy software to WS:

- Manually port original software source code to WS applications. This is an expensive solution. Also WS code, such as WSDL, is verbose, and coding WSDL manually is error prone.

- Language tool based—in which the legacy software package is recompiled to generate WSDL. Many tools such as AXIS[i], and the Microsoft .Net framework provide the function of generating WSDL from implementation code (such as Java and C#) and vice versa. Such tools leverage compiler technology to generate WSDL from other programming languages. The WSDL in turn can be used to generate client side stub code for the client to call the services exposed by legacy software systems (Graham, 2002). However, this language tool based solution remains to be language-dependent. With the variety of legacy software systems, a language neutral solution is required in order to sufficiently handle the reengineering of legacy software systems to WS.

Cao, et al. (2004) used a model-driven approach to WS development. We build upon this work by presenting a model-driven approach for reengineering legacy software systems to the WS applications, in which a model plays a central role for migrating legacy software systems to WS implementations. A model is usually represented in UML[ii], or any other abundant domain specific visual language (as can be seen in JVLC[iii]), which represents the structural and contextual information of a legacy software system in a language neutral style without being tied to implementation specifics. The model-driven reengineering approach is also based on the observation that legacy software systems are usually documented in a visual modeling language; models can also be used as first-class assets in SOA (e.g., model as the basis for service discovery in Hausmann, et al., 2004).

To apply the model-driven approach for reengineering legacy software systems to WS, a model should play a role beyond the conventional design and documentation capacity, i.e., a role for WS code generation directly to resolve the manual porting problem as described above. Usually UML-based code generation is based on a static mapping from the UML profile (Frankel, 2003), which lacks flexibility during code generation process. As such, we use Model IntegratedComputing (MIC) (Lédeczi et al., 2001) for building a WS modeling environment and consequently for WS code generation. MIC is essentially a development paradigm that offers a means for creating a modeling language (*meta-model*), its associated modeling language interpreter (*generator*). Then any domain-specific model built based on the modeling language can be interpreted by traversing the model tree. The result of the interpretation process is the code synthesized from the model. MIC has been widely used in middleware (Gokhale et al., 2004; Edwards et al., 2004) and embedded systems (Karsai et al., 2003; Lédeczi et al., 2003).

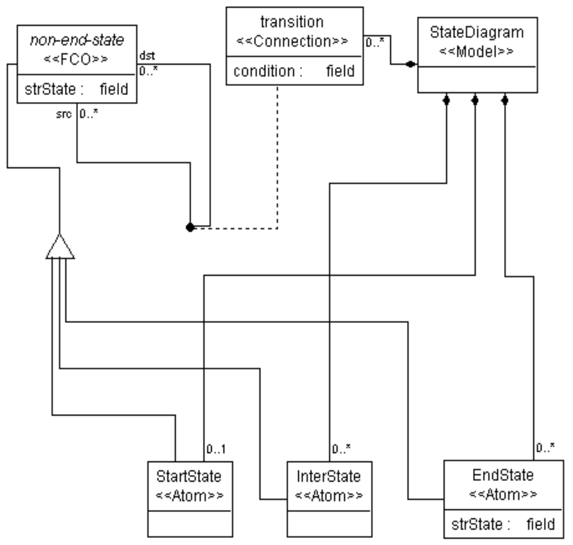
Table 1. Comparison between MIC and programming language

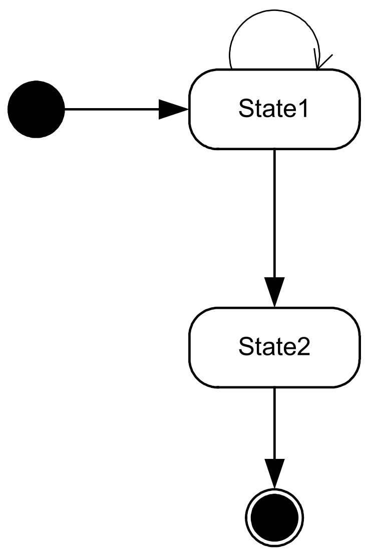
MIC	**Programming Language**
meta-model	grammar
generator	compiler/interpreter
domain-specific model	application developed using the corresponding language
code synthesized in any chosen language	intermediate code or native code

To ease the understanding of MIC, Table 1 provides an analog between MIC and conventional programming language elements. Figure 2 provides an example of a meta-model of Finite State Machine (FSM) and the corresponding model based on it.

While the meta-model (and in the later part the domain-specific modeling environment) described in this paper is based on the notation of the Generic Modeling Environment (GME) (ISIS, 2001) (as it is the only tool for the MIC paradigm so far), the same principle as shown in this paper can be applied to other MIC-compliant modeling tools as well.

Figure 2. A simple example of meta-model and model





Finite State Machine (FSM) Meta-model *Finite State Machine Model*

Problems for Applying Model Integrated Computing (MIC) to Reengineering Legacy Software to WS

While MIC offers an automatable and language neutral approach for reengineering legacy software to WS, the starting point of MIC - the construction of the meta-model has to be a manual process. Previous work on WS modeling (Cao et al., 2003) has revealed that with the increasing complexity of the modeling target, the construction of the meta-model is subject to being ad-hoc and error-prone. With the modeling assets (UML or other domain specific visual modeling language) already abundantly available as part of the legacy software (which we term *legacy model*), it is desirable to derive the meta-model from the legacy model in a systematic, automatable process as opposed to being ad-hoc and error-prone. However, the current meta-modeling languages lack adequate modularity support for large scale meta-model construction, which nevertheless is widely existing in general programming languages. As a result, the construction of a meta-model remains an art rather than a science.

Therefore, this paper is composed of two major parts, each corresponding to the primary contributions of this paper:

1) the elicitation of a meta-model from a legacy model in a systematic, automatable process, which is addressed in Section 2 and Section 3, and consequently
2) the creation of a domain-specific WS modeling environment for WS code generation in Section 4, as well as the treatment of WS semantic concerns from a model-driven perspective in Section 5.

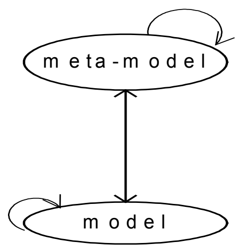
Related work is described in Section 6, followed by the conclusion and future work in Section 7.

MARSHALING AND UNMARSHALING MODELS USING THE ENTITY-RELATIONSHIP (ER) MODEL

The elicitation of a meta-model from UML or other domain-specific modeling notations can be done on a per source model basis. However, with the constant emergence of new modeling notations, the elicitation approaches will become ad-hoc and not reusable. Moreover, there is a need to converge the diversified modeling assets for modeling tool integration[iv]. Therefore, we need to encode the diversified models with a common representation, such that different modeling notations can transfer to and from it, thus modeling assets can be exchanged and used across different modeling tools. Cao et al. (2005) have referred to these modeling notation transferals as *marshaling* and *unmarshaling*, respectively. The term marshaling comes from the distributed computing scenario where heterogeneous data types are always translated into some common data type over the network so as to be consumed at another end of the distributed environment, where the common data type is unmarshaled again into another environment-specific data type. Comparatively, the concept of marshaling and unmarshaling models refers to transform a model to an *intermediate common semantic* form, which is reinterpreted in another modeling environment/tool. This intermediate common semantic form is in a similar vein to

ACME (Garlan et al., 2000), which is an intermediate form for exchanging software architecture description languages across different software architecture design tools. Moreover, with the heterogeneity of models at different meta-level (not only model level but also meta-model level) (Frankel, 2003), marshaling and unmarshaling of models can be performed at different levels: horizontally, meta-model level and model-level; vertically, meta-model to/from model as is illustrated in Figure 3.

Figure 3. Marshaling and unmarshaling models at different levels: the arrow represents marshaling/unmarshaling process



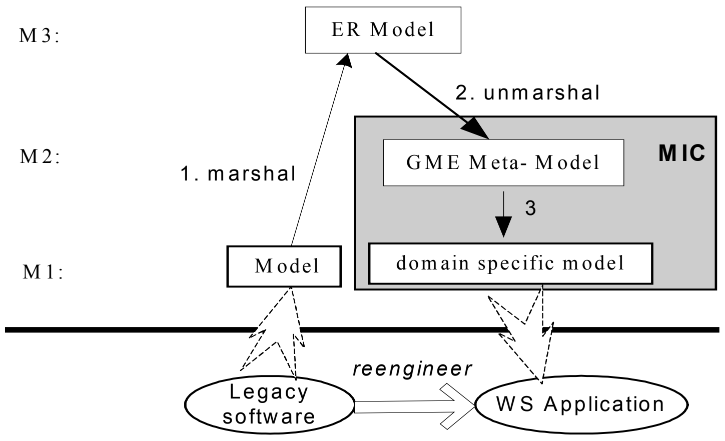
Here we use the ER model (Chen, 1976) as the intermediate common semantic form for marshaling and unmarshaling models[v]. The rationales are as follows:

- **Sufficiency.** Even though UML is widely adopted in software modeling, which seems to justify the use of UML as a common model for exchanging model assets across modeling facilities, UML is not convenient for model serialization, thus not fit for modeling asset exchange, reuse and evolution. In fact, the object diagram (Booch et al., 1999), for which UML is used to capture and store the snapshot of software system state, is represented virtually in an Entity (object) and Relationship (links) model. Moreover, the UML modeling language has its roots in the ER model, and the latter is already widely used as the foundation for CASE tools in software engineering and repository systems in databases[vi].

- **Necessity.** As is illustrated in Figure 3, not only models, but also meta-models are in need of marshaling and unmarshaling. Therefore, the intermediate model should be expressive enough to be at the meta-meta model level in the meta-level stack (Frankel, 2003). The meta-meta-model is described by the Meta Object Facility (MOF)[vii], which is a set of constructs used to define meta-models. The MOF constructs are the *MOF class*, the *MOF attributes* and the *MOF association*. These constructs correspond to an ER representation (by using an Entity to represent a MOF class), which indicates that the ER representation is semantically equivalent to MOF fundamentally. Therefore, the ER representation is the right vehicle to play the dual roles of marshaling both models and meta-models. Also, other non-UML based languages, even though not as popular, are abundantly present, for which UML is not an omnipotent cure.

The scope of this paper is on vertical direction which is further illustrated in Figure 4, i.e., marshaling models to ER model, then unmarshaling ER model to the GME meta-model. The gray area in Figure 4 represents the MIC paradigm. To be specific, in the following section, we will marshal a UML class diagram for Web Services Description Language (WSDL) to the GME meta-model, then create a WS modeling environment based on the meta-model for WS code generation. Therefore, legacy software systems can be reengineered to the WS application automatically with a language neutral approach. We also show the generality of this approach: even though the scope is within the vertical direction, the approach can also be applied for horizontal marshaling/unmarshaling using ER model; even though the source model is the UML object-oriented model, it is not tied to this single kind of source model and can be applied to other domain-specific visual modeling languages as well.

Figure 4. Eliciting Meta-models from model via marshaling and unmarshaling models using ER model



REENGINEERING LEGACY SOFTWARE TO WEB SERVICES (WS)

In order to reengineer legacy software to WS, we need to capture 1) the WS technology domain knowledge; 2) the original legacy software business domain knowledge; and 3) original implementation technology information. This categorization of technology domain knowledge and business domain knowledge has been described by Zhao, et al. (2003).

Figure 5 is the class diagram of WSDL. The WS ***message***s, which are either ***input*** or ***output*** messages, are composed of ***part***s, each of which corresponds to a specific data ***type***. The ***portType*** is an abstract WS interface definition, where each contained element, i.e., the ***operation***, defines an abstract method signature. The operation uses messages as its parameters. ***Binding*** represents an instantiation to the abstract ***portType*** with concrete protocol and data type. ***Service*** is a collection of ***port***s, denoting a deployment of a binding at a specific network location.

Figure 5. The architecture of WS description elements

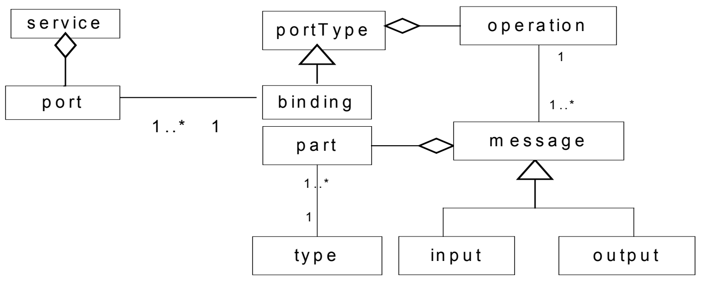


Figure 6 describes the legacy banking application information, including its business domain knowledge (the first two paragraphs) and its original technology domain knowledge (the last paragraph). Note as WS is used as wrapper for original technology domain knowledge together with the business domain knowledge, rather than replacing the original technology, we treat the original domain knowledge as the part of business domain knowledge in the remaining part of the paper for simplicity purpose.

Figure 6. A banking example

A bank provides the service for users to set up accounts. Account information includes personal data including Name, SSN, phone number, address, and account data including Account Number, PIN, Transaction Record, Balance. There are two types of accounts: checking account and savings account.

For the bank side, it provides such services as: Account Verification, Account Query, Deposit, Withdraw, and Transfer.

The banking service implementation may use such technology as RMI[viii], J2EE[ix], and CORBA[x]. Also it will enforce some Quality of Service (QoS) requirements such as Availability, Dependability, Capacity.

Marshaling Legacy Software Model to ER Model

In order to elicit the banking domain WS meta-model, we need to first merge the WS technology domain information with the business domain information. To that end, we treat the WS technology domain as the *dominant domain* during the merge process, with the business domain knowledge as the *adjunct domain* being appended to the marshaled model from the technology domain model. As such, the marshaling process as illustrated in Figure 4 can be decomposed into the marshaling type A for dominant domain and type B for adjunct domain together with a merge step as is illustrated in Figure 7.

Figure 7. Stepwise marshaling

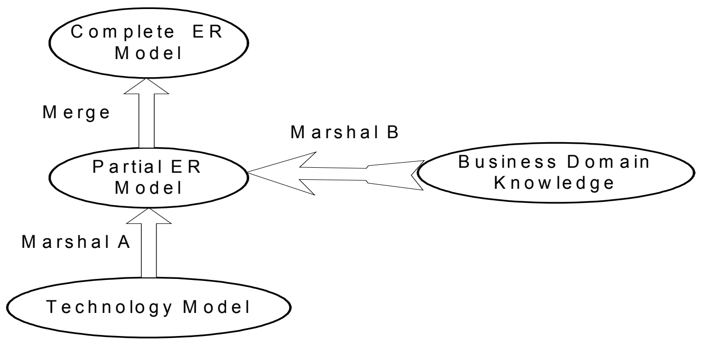
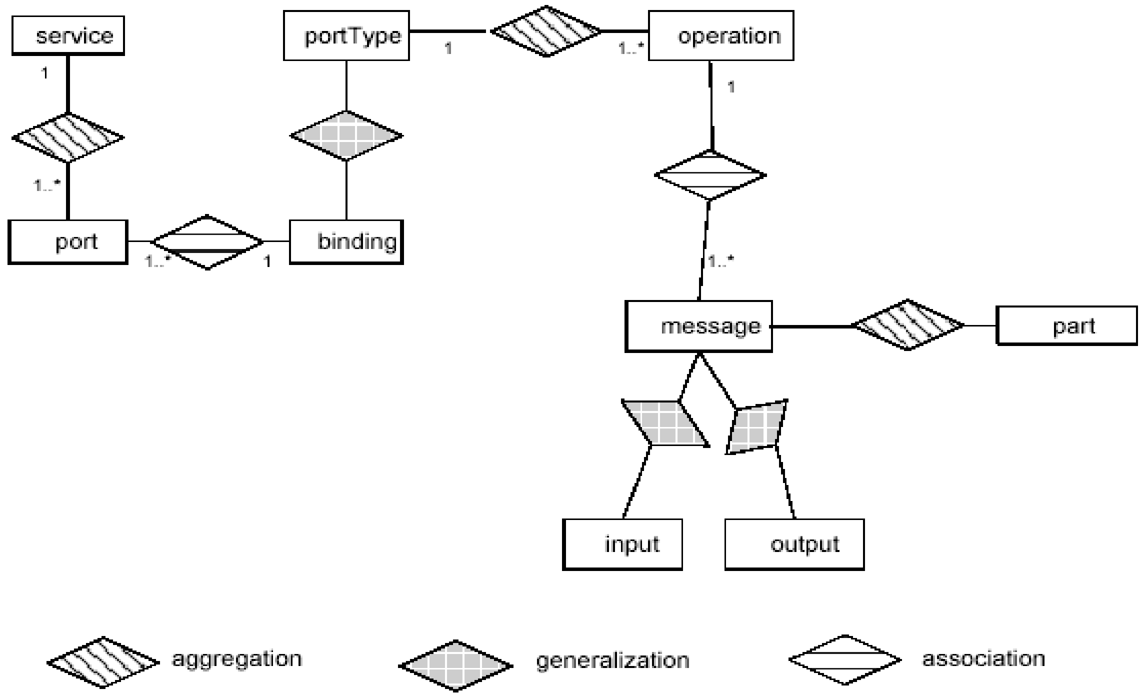


Table 2. Marshaling rules

Type	**Rule**
Marshal A	▪ aggregation, association, generalization, and dependency => *Relationship* ▪ class=> *Entity*
Marshal B	domain analysis and mapping

Table 2 illustrates the marshaling rules based on different marshaling types. Note that one of the essential characteristics of a meta-model is that it treats not only the models, but also the inter-relationships among models as first-class entities. Therefore, for marshal type A, the different type of relationships between classes will be mapped to the *Relationship* construct in the ER model, while each class is represented as an *Entity*. Figure 8 illustrates the resultant ER model after marshaling the WS class diagram based on this rule. Each diamond represents a type of relationship in the original class diagram. Note we ignore *type* in the ER model of Figure 5,

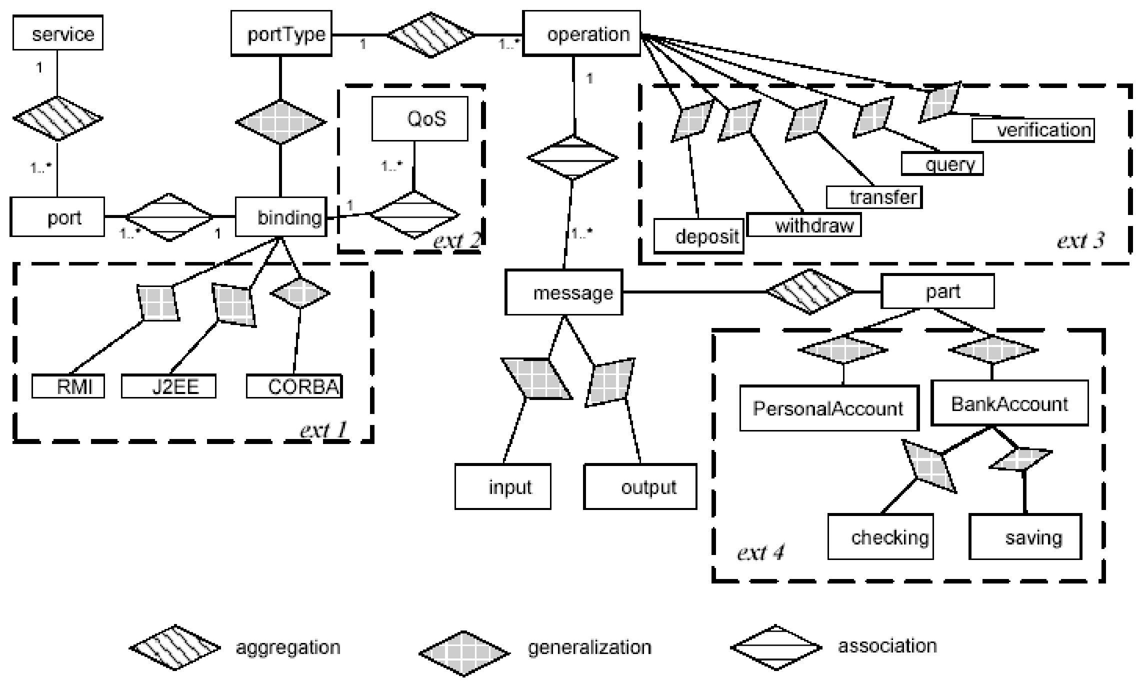
Figure 8. Marshaling WSDL model to ER model



because we can put the type directly as the attribute of the part element. However we will not include the attributes to the entities and relationships in the ER representation here, as the focus of this paper is about the model of marshaling and unmarshaling structurally; the attributes will be annotated in the GME meta-model and are shown later.

For marshal type B, a domain analysis phase (Czarnecki & Eisenecker, 2000) is needed to associate the business domain information to the technology domain information. Specifically, the different banking services described in Figure 6 can be treated as different types of **operations** in WSDL, while different banking service implementation technology and QoS requirements can be associated to **bindings** in WSDL as a reification of operations. Account information and account type information can be treated as **messages** in WSDL. Figure 9 illustrates in detail the resultant ER model after annotating the business domain knowledge (using either generation relationship or association relationship) to the WSDL ER model illustrated in Figure 8. By using the ER model as the intermediate form for marshaling, different types of

Figure 9. The ER model of Banking Service WSDL: the three parts enclosed with dashed line represent the extended part to the WSDL model.



domain knowledge can be merged incrementally without obfuscating each other, which provides a separation of concerns toward domain-specific model refinement. Also with the non-invasive merge process, the business domain semantics are reified with technology semantics while the business domain semantics are kept unchanged.

Just as the compiler can apply code optimization when compiling application code, the marshaling process can be used to apply optimization (e.g., reduce redundant models or relationships) for the original modeling language (either UML or domain specific), the detailed discussion of which is out of the scope of this paper.

Unmarshaling ER Model to GME Meta-model

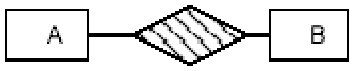
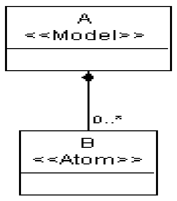
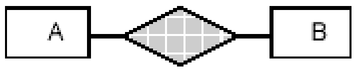
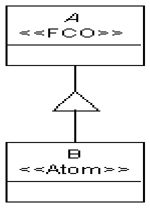
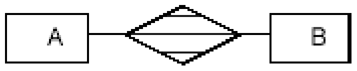
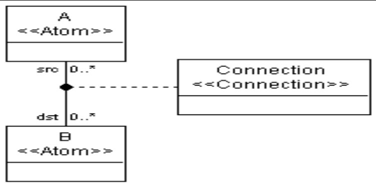
In the GME meta-model, the containment relationship is represented by using a *model* element (stereotyped with <<*model*>>), which, in contrast to an *atom* element (stereotyped with <<*atom*>>), can contain other modeling elements. Also the contained elements can be promoted as *ports* of the model to have direct connections with external modeling elements. Additionally, GME uses a *root model* as an entry point of access to all the modeling elements. Also, the *relationship* of ER is represented in GME as a first-class modeling element, *connection* (stereotyped with <<*connection*>>), with a *connector* in the form of a dot to associate this relationship with two modeling elements (entities).

The unmarshaling from the ER model to the GME meta-model is based on the relationships in the ER representation, as is illustrated in Table 3.

1) A contains B. In this case, A can be modeled as a *model* element in GME containing B.

2) B is specialized from A. In this case, A is rendered by an abstract FCO (First Class Object, tagged with <<*FCO*>>, represents an abstract generalization of other modeling constructs), a modeling element to be used as an abstract interface in GME, and B is represented as an inherited class of that FCO. Note there are two special treatments here: first, for the input/output elements

Table 3. The Unmarshaling Rules: the relation notation is consistent with that in Figure 8

Rule Number	Relationship type	GME Metamodel element
1		
2		
3		 A <<Atom>> src 0..* Connection <<Connection>> dst 0..* B <<Atom>>

of Figure 9, they are only used to tag the *connection* (named either “input” or “output”) between message entities and its interconnecting entities in GME; second, the generalization relationship between binding and portType is actually treated as an association when modeling in GME, because the binding entity actually attaches values of the chosen protocol to the portType in WSDL rather than in the real sense of inheritance.

3) B is associated to A. In this case, a *connection* can be added to be associated with the A and B representations in GME. The connection element can be named with respect to A’s or B’s properties as a kind of tag, e.g., the tag can be named as the combination of both A’s name and B’s name. Note when the situation as described in case 2 applies, then this tag should be named as in case 2.

Figure 10. The meta-model of banking domain WSDL in GME

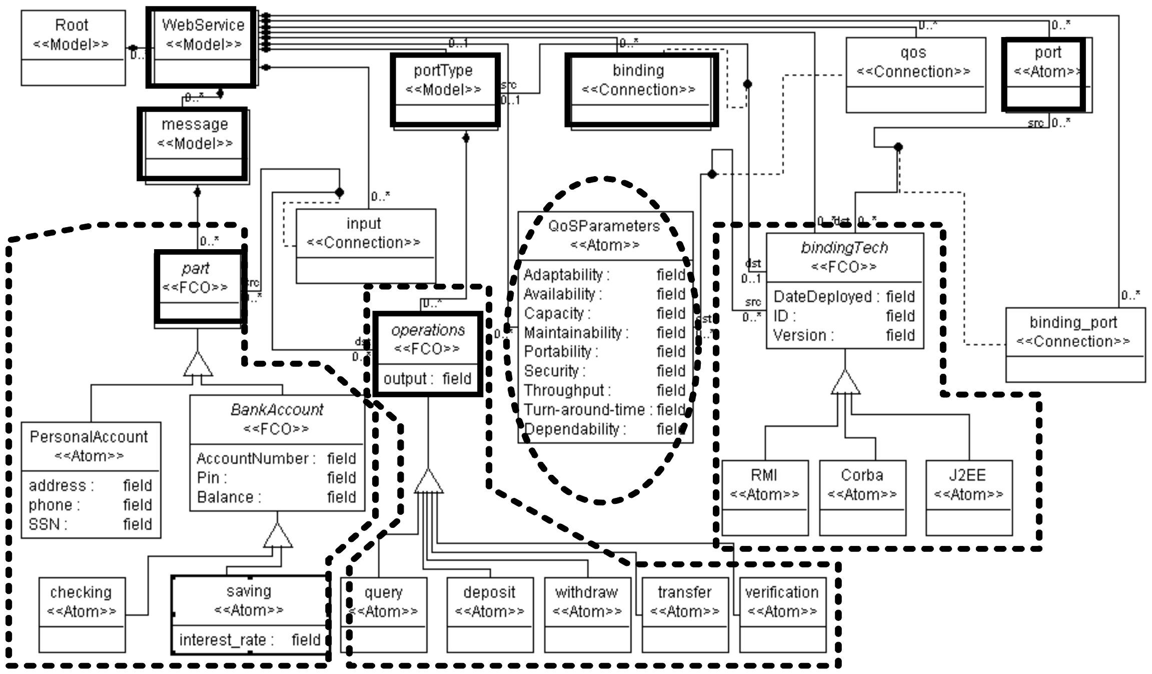
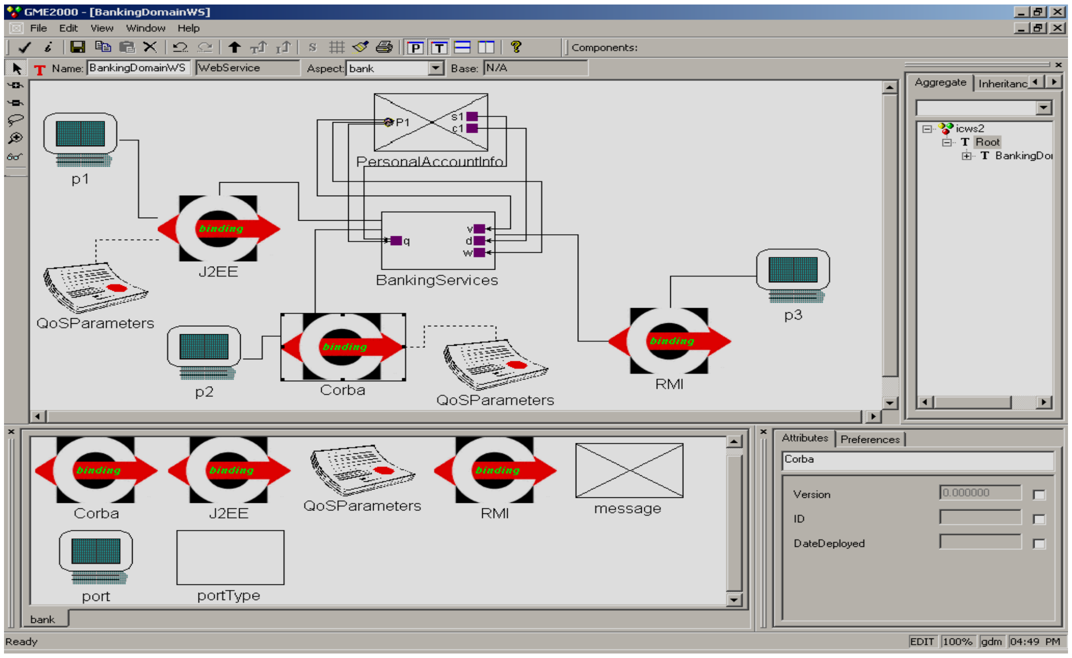


Figure 10 shows the meta-model created by unmarshaling the ER model in Figure 9 strictly observing the above unmarshaling rules. The seven boxes with bold borders correspond to the seven WSDL entities in Figure 8 and 9, with *WebService* corresponds to the *service* entity. The boxes in Figure 10 also contain attributes for the related models to be instantiated in the modeling phase. The four areas designated by four bold dashed circular lines correspond (from right to left) to the extension parts 1-4 in Figure 10. It can be seen from Figure 10 that the meta-modeling language lacks the modularity that programming languages have, thus the construction process of a complex meta-model is error-prone without a systematic, automatable treatment.

THE WS MODELING ENVIRONMENT

After a meta-model is derived by marshaling and unmarshaling models, a domain specific modeling environment (which is also a crucial part of MIC) can be created based upon the meta-model, as is indicated in Table 1. Figure 11 shows the screenshot of the banking-domain WS modeling environment based on the meta-model illustrated in Figure 10. The lower-left corner provides the modeling elements that can be dragged and dropped in the upper-left pane for

Figure 11. The banking domain-specific WS modeling environment



constructing a banking service model. The names of the models in the lower-left pane represent the meta-model names (*kind names*); when those models are dragged to the above pane, the model name can be changed to reflect the meaning of the model in the domain-specific context, which we call a *context name*. Furthermore, the domain-specific model can be traversed based on the meta-model and interpreted in terms of code generation using the GME Builder Object Network (BON) framework (ISIS, 2001), which is illustrated in Figure 12. For saving space, Figure 12 only shows the interpreter code for generating message and portType of WSDL. Other part of WSDL can be generated in a similar way. The WSDL code generated for the banking service embedded with QoS parameter extension is shown in Figure 13. Because of the limited space, only a snippet of the generated WSDL code is shown in Figure 13. Notice the bold-font part of the following WSDL code includes the QoS and ontology attributes of WSDL, which may be used for WS filtering if QoS requirements or domain specific requirements are include for service discovery.

Figure 12. WSDL code synthesis using GME BON API

```
const CBuilderModelList *root = builder.GetRootFolder()->GetRootModels();
POSITION pos = root->GetHeadPosition();
ASSERT(pos->GetCount()==1);  //to ensure this model is representing just one WSDL

CBuilderModel *webserv = pos->GetHead(); //get the handle to the WebService model
ASSERT(webserv->GetKindName()=="WebService");

//WSDL message part
const CBuilderAtomList *messages = webserv->GetModels("message");
pos=messages->GetHeadPosition();
CBuilderAtom *oneMessage;
while(pos)
  {
    /*
     traverse each message model and generating code
     <message>... </message>
     for each message model
    */

     oneMessage=messages->GetNext(pos);
     const CBuilderAtomList *accounts =oneMessage->GetAtoms("PersonalAccount");
    ...
   }

//WSDL portType part
const CBuilderAtomList *portType = webserv->GetModels("portType");
pos=portType->GetHeadPosition();
ASSERT(pos->GetCount()==1);  //to ensure only one portType element in WSDL
CBuilderAtom *oneportType;
oneportType=portType->GetNext(pos);
…..
}
```

Figure 13. The WSDL for a banking WS

```
<message name="checking">
 <part name="user_ident" type="identity"/>
 <part name="p1" type="checking"/>
</message>
<message name="savings">
 <part name="user_ident" type="identity"/>
 <part name="p1" type="savings"/>
</message>
<message name="checking_savings">
 <part name="user_ident" type="identity"/>
 <part name="p1" type="checking"/>
 <part name="p2" type="savings"/>
</message>

<portType name="BankingServices">
    <operation name="w"
            ontology="Banking:withdraw">
      <input message="checking"/>
      <output message=""/>
    </operation>
   <operation name="d"
            ontology="Banking:deposit">
       <input message="checking"/>
       <output message=""/>
   </operation>
  <operation name="v"
            ontology="Banking:deposit">
   <input message="checking_savings"/>
   <output message=""/>
  </operation>
  <operation name="q" ontology="Banking:query">
     <input message="savings"/>
      <output message=""/>
  </operation>
</portType>
```

(to be continued in the right pane)

```
<binding name="J2EE_Banking"
            type="BankingServices">
  <soap:binding style="J2EE" transport="http"
    QoS:portability="0.544400">
    .........
</binding>
<binding name="CORBA_Banking"
            type="BankingServices">
  <soap:binding style="CORBA" transport="IIOP"
    QoS:turn-around-time="10.35">
    .........
</binding>
<binding name="RMI_Banking"
            type="BankingServices">
  <soap:binding style="RMI" transport="http"
    QoS:dependability="0.34">
    .........
</binding>

<service name="My Bank">
  <port name="p1" binding="J2EE_Banking">
      <soap:address location="URL1"/>
  </port>
  <port name="p2" binding="CORBA_Banking">
      <soap:address location="URL2"/>
  </port>
  <port name="p3" binding="RMI_Banking">
      <soap:address location="URL3"/>
  </port>
</service>
```

MODEL-DRIVEN APPROACH TO ENRICH WS SEMANTICS

Current WS standards mainly embrace the semantics of processes at the collaborating syntactic interface level. WSDL only exposes distributed object services, while such process behavior aspects as ordering, and dependency are not well specified in the existing WSDL standard. The model-driven approach can play a unique role in enriching the WS semantics:

- OCL (Object Constraint Language)[xi] to enrich WS semantics at a high level

  OCL is used to complement the semantic representation for UML. Likewise, when the model is used to represent WS, OCL can be used to enrich WS semantics indirectly at a higher level. For example, if we add into the banking case in Figure 6 such requirement that "deposit and withdraw can only be applied to checking account", the specified constraints over withdraw and deposit operations can be enforced in GME using the following MCL expression (ISIS, 2001), an OCL implementation in GME:

  ```
  connectedFCOs("src")->forAll(c|c. kindName()="checking")
  ```

  Those constraints apply to both the withdraw atom and the deposit atom in Figure 10, which means those First Class Objects (referring to both entities and relations in GME) that are connected with withdraw/deposit atoms are all of kind "checking". Therefore, in the WS modeling environment as shown in Figure 11, once a modeling entity of type other than "checking" is connected to withdraw/deposit, an error message window will pop up.

- Meta-model as Ontology

  A valid meta-model is an ontology, but not all ontologies are modeled explicitly as meta-models (Ernst, 2002). This ideal has already been used in (Hausmann et al., 2004) for WS discovery. Comparatively, here we just output the meta-model information into the generated WSDL as ontology annotation to enrich the WSDL semantic representation.

- Creating modeling language for enriching WS semantics

Assume there is order restriction for those banking operations described in Figure 6: both *transfer* and *withdraw* have to be preceded by a *query* operation; the *account verification* comes after each of the other operations. Such models as Finite State Machine (FSM) can be used to enrich WS semantics. Based on the FSM meta-model in Figure 2, a FSM modeling environment can be created in addition to the WS modeling environment that is described in Section 4, which can be used to generate operation ordering constraint code to be embedded in WSDL. We skip the details here due to space limitations.

RELATED WORK

This paper presents both a novel model-driven approach in general and its novel application to WS in particular. Specifically:

1) For the model-driven approach aspect, we use ER model for marshaling and unmarshaling models. The related work in this regard includes:

- **MDA**

  MDA[xii] is an initiative from OMG[xiii] for capturing the essence of a software system in a manner that is independent of the underlying implementation platform. MDA can assist in reengineering legacy software systems into Platform Independent Models (PIMs). A PIM can be mapped to software components on Platform Specific Models (PSMs), such as CORBA, J2EE or .NET. In this way, legacy systems can be reintegrated into new platforms efficiently and cost-effectively (Frankel, 2003). However, the core part of mapping technology for MDA is either ad-hoc or pre-mature before MDA can be fully adopted in industry. ER-based model marshaling and unmarshaling offers a potential solution to address this problem systematically. Another difference is that in MDA, the PIM is treated as dominant model while here we treat the technology domain as dominant model, with business domain knowledge (PIM) as adjunct model in Section 3.

It has been observed that the ER representation has been adopted in defining the Knowledge Discovery Meta-Model (KDM)[xiv] and Ontology Definition Meta-Model (ODM)[xv] in OMG, which underscores the role that ER plays for model marshaling and unmarshaling.

- **Grammar Inference**

  The ER model, because of its powerful modeling capacity, can be used as an intermediate form for model-to-model and meta-model-to-meta-model exchange. Because of the dual role that the ER model can play, it is treated as an intermediate form for model-to-meta-model elicitation, which is the theme of this paper. This idea is very similar to grammar inference (Higuera, 2001), where a grammar can be inferred from language examples. But the two approaches are applied at different abstraction levels.

- **XMI**

  XMI[xvi] provides a standard mapping from MOF-based models to XML, which can be exchanged between software applications and tools, and the XMI specification is difficult to read by humans. In contrast, ER-based model marshaling and unmarshaling represents a design-level approach for evolving design assets, without being restricted to low-level syntactical data representation specifics, and the ER representation is much more human comprehensible. Also, the XMI-based approach uses top-down mapping, and is coupled to the meta-model of the targeted language; interchange format cannot be changed without changing the meta-model. In contrast, the ER-based approach represents either horizontal mapping or bottom-up mapping as is illustrated in Figure 3, without being tied to any meta-model.

2) We applied the model-driven approach to WS, specifically, MIC for WS code generation automatically; Model-driven approaches for enriching WS semantics are also identified. The related work in this regard is as follows:

In Lopes and Hammoudi (2003), MDA is used together with workflow technology for modeling and composing WS. But the authors do not provide a guideline as to how to create the meta-models. Also the mapping from PIM to PSM is not detailed. In contrast, our meta-modeling approach is sufficiently complete and general as to be applicable to other aspects of WS such as WS orchestration code generation. Sivashanmugam (2003) describes an approach of adding semantics to WS by adding ontology attributes to both WSDL and UDDI, which includes pre-condition and effect specification. We applied ontology annotation to WS as well, and we put the pre-condition and other effect specification at the meta-model level. In Mantell (2003), an MDA approach is used for BPEL4WS[xvii] code generation from a UML design. This approach uses XMI processing technology for UML model exchange. Comparatively, the XML representation for the ER model is much simpler and easier to process in our approach. Code generation in Mantell (2003) is based on the UML profile mapping, which is not as flexible as a generator-based approach in our case.

The UniFrame project (Raje et al., 2002; Olson et al., 2004), has a more comprehensive application of the model-driven approach. UniFrame aims at creating a framework for seamless integration of distributed heterogeneous components. In UniFrame, the model-driven approach is applied for domain engineering, and for creation of Generative Domain Models (GDMs) (Czarnecki and Eisenecker, 2000), which are used for eliciting rules to generate glue/wrapper code for assembling distributed heterogeneous components. In contrast, the scope of glue/wrapper code generated here is specific to WS code, which has not been addressed by UniFrame.

CONCLUSION AND FUTURE WORK

With Web Services (WS) as a wrapper, legacy software systems can be reused and integrated beyond enterprise boundaries across heterogeneous platforms. This paper explores in detail a model-driven approach to reengineer legacy software system to WS applications using a systematic, automatable process, which includes: 1) the meta-modeling process using ER-based marshaling and unmarshaling, 2) the construction of a WS modeling environment for generating

WS code and enriching WS semantics. To our best knowledge, there is no peer work that addresses either systematic meta-model construction, or sufficient model-based WS code generation, while our work represents a comprehensive solution to both issues. Even though the work presented in this paper is specific to WS development, the approach can be applied to other web system engineering by reengineering to a different meta-model other than the WS meta-model.

Future work will be to provide tool support for part 1 in the preceding paragraph to automate the model marshaling and unmarshaling process for seamlessly integrating the reengineering process to MIC paradigm. For part 2, we will enrich the WS modeling environment by providing modeling and code generation support to other behavior concerns of WS such as interaction, activity, and temporal relationship, as well as WS orchestration and adaptation.

ACKNOWLEDGEMENTS

This research is supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

REFERENCES

Booch, G., Rumbaugh, J. & Jacobson, I.(1999). The Unified Modeling Language User Guide. Addison-Wesley.

Cao, F., Bryant, B. R., Burt, C., Gray, J., Raje, R., Olson, A., & Auguston, M. (2003). Modeling Web Services: toward system integration in UniFrame. *Proceedings of 7th World Conference on Integrated Design and Process Technology (IDPT'03)*.

Cao, F., Bryant , B. R., Zhao, W., Burt, C., Gray, J., Raje, R., Olson, A., & Auguston, M. (2004). A Meta-modeling approach to Web Services. *Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004).*

Cao, F., Bryant , B. R., Zhao, W., Burt, C., Gray, J., Raje, R., Olson, A., & Auguston, M. (2005). Marshaling and unmarshaling models using Entity-Relationship model. *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC 2005)*.

Chen, P. P. (1976). The Entity-Relationship model: toward a unified view of data. ACM Transactions on Database Systems, 1(1), 9-36.

Colan, M. (2004) Service-oriented architecture expands the vision of Web Services. http://www-106.ibm.com/developerworks/webservices/library/ws-soaintro.html.

Czarnecki, K., & Eisenecker, U.W. (2000). Generative Programming: Methods, Tools, and Applications. Addison Wesley.

Devanbu, P., Karstu, S., Melo, W., & Thomas, W. (1996). Analytical and empirical evaluation of software reuse metrics. *Proceedings of 18th International Conference on Software Engineering (ICSE'96)*.

Edwards, G. T., Deng, G., Schmidt, D. C., Gokhale, A. S., & Natarajan, B. (2004). Model-driven configuration and deployment of component middleware publish/subscribe services. *Proceedings of 3rd international Conference on Generative Programming and Component Engineering (GPCE 2004).*

Ernst, J. (2002). What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model? http://www.metamodel.com/article.php?story=20030115211223271.

Frankel , D. S. (2003). Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley.

Garlan, D., Monroe,R. T., & Wile, D. (2000). Acme: architectural description of component-based Systems. Foundations of Component-Based Systems, ed. Leavens, G. T. and Sitaraman, M., Cambridge University Press, 47-68.

Gokhale, A., Schmidt, D. C., Natarajan, B., Gray, J., & Wang, N. (2004) Model driven middleware. Middleware for Communications, ed. Mahmoud, Q., John Wiley and Sons, 163-187.

Graham, S., Simeonov, S., Boubez, T., Davis, D., Daniels, G., Nakamura, Y. & Neyama, R. (2002). Building Web Services with Java. SAMS.

Hausmann, J. H., Heckel, R., & Lohmann, M. (2004). Model-based discovery of Web Services. *Proceedings of International Conference on Web Services (ICWS 2004).*

Higuera, C. d. l. (2000). Current trends in grammatical inference. *Proceedings of Joint IAPR Int. Workshops SSPR & SPR 2000*.

ISIS.(2001). GME 2000 User's Manual, Version 2.0. Vanderbilt University.

Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T. (2003). Model-integrated development of embedded software. IEEE. 91(1), 145-164.

Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing domain-specific design environments. IEEE Computer, 34(11), 44-51.

Lédeczi, Á., Davis, J., Neema, S., Agrawal, A. (2003). Modeling methodology for Integrated simulation of embedded systems, ACM Transactions on Modeling and Computer Simulation. 13(1), 82-103.

Lopes, D., & Hammoudi, S. (2003). Web service in the context of MDA. *Proceedings of. International Conference on Web Services (ICWS'03).*

Mantell, K. (2003). From UML to BPEL: model driven architecture in a Web Services world. http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel/.

Olson, A. M., Raje, R. R., Bryant, B. R., Burt, C. C., & Auguston, M. (2004). UniFrame-a unified framework for developing service-oriented, component-based, distributed software systems. Service-Oriented Software System Engineering: Challenges and Practices, ed. Stojanovic, Z. and Dahanayake, A., Idea Group, 68-87.

Raje, R. R., Auguston, M, Bryant, B. R., Olson, A. M., Burt, & C. C. (2002). A quality of service-based framework for creating distributed heterogeneous software components. Concurrency and Computation: Practice and Experience, 14(12), 1009-1034.

Sivashanmugam, K., Verma, K., Sheth, A., & Miller, J. (2003). Adding Semantics to Web Services Standards. *Proceedings of International Conference on Web Services (ICWS'03).*

Zhao, W., Bryant, B. R., Burt , C. C., Gray, J. G., Raje, R. R., Olson, A. M., & Auguston, M.(2003). A generative and model driven framework for automated software product generation. *Proceedings of CBSE 6, the 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*.

[i] http://ws.apache.org/axis/
[ii] UML^{TM} - Unified Modeling Language - http://www.omg.org/uml
[iii] JVLC - Journal of Visual Languages and Computing-http://www.elsevier. com/locate/jvlc
[iv] Interview with Keith Short, http://www.theserverside.net/talks/ library.tss#KeithShort
[v] Note that the ER model is not intended to replace the existing modeling language such as UML or Petri Nets – those modeling languages have their own advanced features for a specific domain to model. Here the ER model is chosen as an intermediate form only for exchanging models of a close type or serving a close purpose but with variant notations across different modeling tools and environments.
[vi] http://bit.csc.lsu.edu/~chen/chen.html
[vii] Meta-Object Facility - http://www.omg.org/technology/documents/formal/mof.htm
[viii] RMI - Remote Method Invocation: http://java.sun.com/products/jdk/rmi/index.jsp
[ix] J2EE - Java 2 Enterprise Edition: http://java.sun.com/j2ee/
[x] CORBA® - Common Object Request Broker Architecture: http://www.omg.org/corba/
[xi] http://www-3.ibm.com/software/ad/library/standards/ocl.html
[xii] MDA - Model-Driven Architecture - http://www.omg.org/mda
[xiii] OMG - Object Management Group -http://www.omg.org/
[xiv] http://www.omg.org/cgi-bin/doc?lt/2003-11-4
[xv] http://codip.grci.com/odm/draft/submission_text/ODMPrelimSubAug04R1.pdf
[xvi] XMI - XML Metadata Interchange - http://www.omg.org/technology/ documents/formal/xmi.htm
[xvii] BPEL4WS - Business Process Execution Language for Web Services - http://www-128.ibm.com/ developerworks/library/specification/ws-bpel