UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages

Endre Magyari, Arpad Bakay, Andras Lang, Tamas Paka, Attila Vizhanyo, Aditya Agarwal, and Gabor Karsai

Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Domain-specific modeling languages amortize the cost of the development of a language over all the software products they can be used for. This paper describes an infrastructure for developing DSMLs and using them in a systematic manner. The infrastructure consists of a modeling language (UML class diagrams), a modeling tool, a code generator, and a number of generic libraries that provide support for object performance in several forms.

Introduction

Domain-specific modeling languages (DSML) allow domain-oriented software development, where *models* (expressed using the DSML) capture the variabilities present in the various applications of a product line and generators are used to produce the code that implements those variabilities. The cost of the development in this case includes the cost of developing a DSML, and then using it for building all the elements of a product-line, hence any improvements in the DSML development can greatly reduce the overall cost.

Rapid definition and implementation of DSML-s necessitates the construction of a data access and manipulation layer that forms the “backbone” that model editing, database, and the generator tools will use. This data layer should be designed in accordance with the abstract syntax of the DSML, and should provide an interface to some underlying, possibly persistent data storage mechanism. The data layer should also follow some well-formedness rules for the data, such that data structures instantiated satisfy integrity constraints specified in the abstract syntax of the DSML. Furthermore, the data layer should provide services for parsing and unparsing the (textual or visual) DSML, as well as support for converting the data structures of the abstract syntax into problem-specific textual form (e.g. instantiated C code templates). Additionally, to support generic (non-language-specific) tools, the data layer must be reflective; i.e. it should allow access to the meta-data of the DSML. The data layer also must satisfy efficiency requirements; although it is understood that one trades off generality for performance, but the penalty paid should be acceptable.

We have designed and developed an infrastructure: Unified Data Model (UDM) framework that facilitates the construction of DSML-s and provides services as described above. In this paper we describe the design considerations used in implementing UDM, explain how its layers were constructed, introduce the supporting tools, and discusses applications where it was used.

Design considerations for UDM

The design of the UDM framework was heavily influenced by the following objectives.

1. *Use a UML-based tool for specifying the data model.* The first step in using the UDM framework is to describe the data model in a concise, preferably visual way. UDM relies on GME: a generic modeling tool [2], and GME’s support for the UML class diagrams, called the GME/UML paradigm.
   The GME/UML paradigm implements a subset of the OMG UML notation guide[5]. The supported features include packages, class-diagrams, inheritance, composition, simple- and association-class based association, scalar, array, volatile (non-persistent), and ordered attributes, and four basic types for scalar values. Cardinality information can be specified for the compositions and association relationships, as well as for attributes. The syntax for the cardinality and attribute specifiers complies with the OMG UML notation guide.
2. *Provide support for UML metadata interchange (XMI).* The UDM framework defines its own format to represent UML metadata, which is the intermediate file between the GME UML modeling environment and the code-generator that produces code for the interfaces. In essence, this is a concise, XML representation of the UML class diagram.

OMG has also defined the XML Metadata Interchange (XMI) specification [6] to facilitate the exchange of UML metainformation between UML modeling tools and UML metadata consumers (like code generators).
It is important to understand that the UDM metadata is also a UDM model, when the metamodel is the UML meta-meta model.
As the UDM-based metadata representation format serves the same purpose as the XMI specification, and whereas both are serialized using an XML syntax, the UDM framework provides a conversion tool to facilitate interoperability. This allows

1) UDM metadata be exported in XMI format, for use as input to XMI-based tools, such as code generators, and
2) XMI metadata exported by other UML modeling tools to serve as input for the UDM framework. Moreover, UDM API can be generated directly from XMI metadata.

3. *Provide object-oriented C++ interfaces that support convenient, programmatic access.* The object-oriented approach is followed to describe the data structures in the form of UML class diagrams. Note that in UDM we are focusing on pure data, thus all methods of class objects are related to data access and manipulation.
   Convenient access methods are generated for object creation/removal, link creation/removal, and attribute setters/getters. The arguments as well as the return types of the access methods are strictly typed, as implied by the class-diagram. Internally, run-time type checking is strictly enforced in the UDM framework.
4. *Provide generic C++ UDM libraries for implementing the data storage*. The UDM framework provides a set of static libraries, which are generic and can be used with any metadata, and are independent of specific class diagrams, even though the access methods of classes have strictly typed prototypes. This is achieved through generic, inlined template classes that are instantiated in the generated API. The biggest benefit of this generative approach is that the UDM libraries are generic and suitable for any UML metamodel; they don't have to be recompiled for a specific UML metamodel. Also a great benefit is that a UDM application linked with the UDM libraries is able to access models without having to contain any generated (domain-specific) code. However, in such cases the API is not available and access to the model is through a generic, non-domain-specific API. The metamodel of UML (as a DSML) must also be available for loading it runtime, as an instance of the UML meta-meta model. This is used in what we call *dynamic metamodel loading*.
5. *Offer a rich feature-set including support for creating subtypes and instances from instances.* Beyond the normal UML notions like inheritance, composition, association, attributes etc., the UDM framework also supports an inheritance mechanism at the level of instances. This means that an object can be subtyped or instantiated. The subtyped or instantiated object "inherits" all the properties from its archetype: the children, attribute values, and associations. It also remains synchronized to its archetype: whenever the archetype is changed, the subtyped/instantiated objects are also changed. The subtyped objects could be modified independently, while the instantiated objects cannot be modified independently, only through their archetype.
6. *Multiple storage technologies.* The UDM libraries called "backends" provide persistence services for the data structures described in the UML class diagram. Multiple storage technologies are available: the XML/DOM backend facilitates the data exchange with other tools via XML files, the MGA backend uses the native API of GME to access data within GME model databases, and finally the MEM/Static backend uses memory data structures and compact binary files for persistence.
7. *Platform-independence.* The UDM libraries, headers and tools are platform-independent, and currently there are releases for Win32 and Linux platforms.

The UDM Tools and Architecture

The UDM (Universal Data Model) framework defines a development process and a set of supporting tools that are used to generate C++ programmatic interfaces from UML class diagrams of data structures. These interfaces and the underlying libraries provide convenient programmatic access and automatically configured persistence services for data structures as described in the input UML diagram.

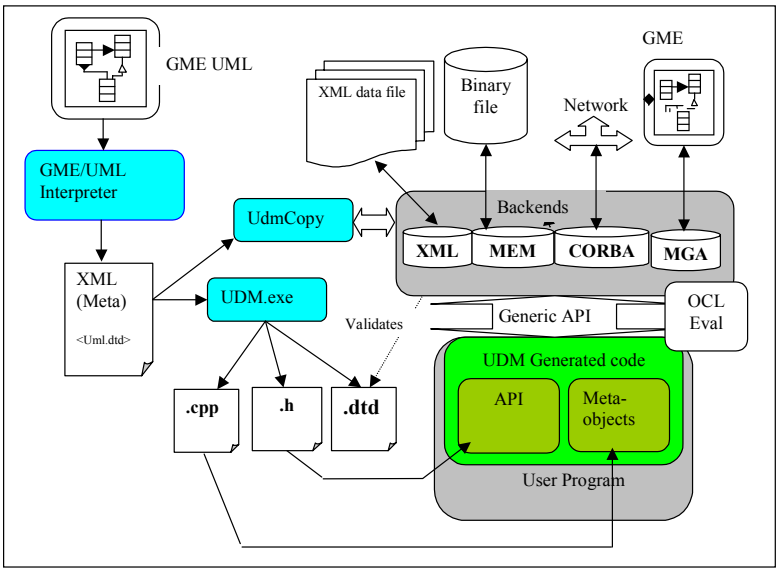

Figure 1 – UDM tools and architecture

The first step in using the UDM framework is to create a UML class diagram in the GME/UML modeling environment. The second step is to interpret the class diagram by using a GME model interpreter. This interpreter module walks the class-diagram, and generates a corresponding XML description of the UML class-diagram – also referred to as 'metainformation' or 'metamodel'. The third step is to generate the API for the class-diagram; this is done by the Udm.exe tool: essentially a code generator. This tool reads the input XML and generates corresponding **.h**, **.cpp**, and **.dtd** files. The **.h** file will contain the generated API (class definitions), while the **.cpp** file contains a function, which initializes (at the first time when it is needed) the meta-objects – the static members of the classes in the generated API.

The next step is to create an application, which includes the generated API and compiles in the application the generated **.cpp** file. The application is typically linked with a generic library ("MEM") that contains the implementation of user–defined objects as simple memory blocks. Such an application is be able to open a data network (containing the objects), create/access objects using the generated API in memory, and also serialize the container and its content to/from a compact, binary file. In order to use the other backends (XML, MGA) additional macro definitions need to be added to the source after including the generated **.h** file and the applications needs to be linked against additional libraries. When using the MGA backend, MGA files are created which can be opened directly in GME [2] and the objects can be viewed visually – objects with hierarchy can be opened as diagrams, links between objects can be seen, etc. When using the XML backend, XML files are created which can be exchanged with other XML-based applications. The DTD file generated by Udm.exe is needed to create new or parse existing XML files. This ensures that the data in the XML file conforms to the metamodel (i.e. the UML class-diagram).

Another binary utility, UdmCopy.exe can be used to copy a data network to another data network. The source and the destination data networks can reside on different backends (MEM to GME, XML to MEM, etc.). This utility also requires the XML file holding the metainformation for the source data network.

At the heart of the UDM libraries there are two pure virtual classes: *ObjectImpl* and *DataNetwork,* which define the abstract interface to access and manipulate objects in a generic way in the backends. The backend libraries then implement the generic interface and thus provide persistency services.

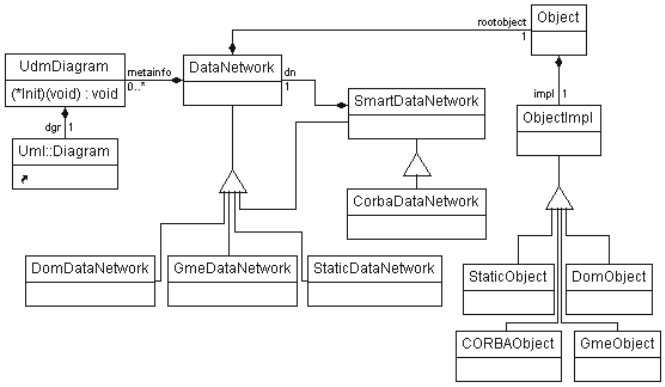

Figure 2 - Core classes

All interactions between the generated API-s and the different backends can only occur through the methods of these two abstract classes. The *Object* class acts like a smart-pointer on *ObjectImpl* – it holds a reference to a real object and implements the proxy pattern and a reference counting mechanism. Objects are contained in data network containers. The *Object* class is the base class for all the classes in the generated APIs. The main idea, which keeps the whole framework relatively simple, is that all the objects - regardless of where they are in the conceptual hierarchy – should be accessed in the same manner. All the objects – the meta-meta model objects (*Uml::Class, Uml::Diagram*, etc.), the UML diagram(meta model – models of DSMLs from which generation occurs) objects and the data objects(objects created in DSMLs) - are all instances of a class which implements the *ObjectImpl* pure virtual class. The meta-meta model is created via a UDM-generated API (Uml.h, Uml.cpp), from a small class-diagram which recursively defines the basic UML notions (*Uml::Diagram, Uml::Class, Uml::Attribute* , etc.), i.e. the metamodel of UML itself, as shown on Figure 3. Since the above metamodel of UML is capable of describing itself, it is possible to use this meta-meta model as a metamodel.

On the other hand, the *ObjectImpl* pure virtual class requires that all implementations of this class should hold a type information – which is an instance of the *Uml::Class* class. This leads to an interesting recursion: an instance of *ObjectImpl* contains an instance *Uml::Class* class – which is derived from *Udm::Object* which contains an instance of *Udm::ObjectImpl.* The tail of this recursion is at the *Uml::Class* meta-object, which has itself as type information.

API generator

The UML description of the data structure is translated into C++ class definitions that define an API that is convenient for the programmer, and gives access to all components of the data structure. For each (abstract or concrete) *UML class* in the source diagram, a C++ class is defined with the corresponding name. All the classes belong to a namespace, which is (by default) named after the package that the class diagram belongs to.

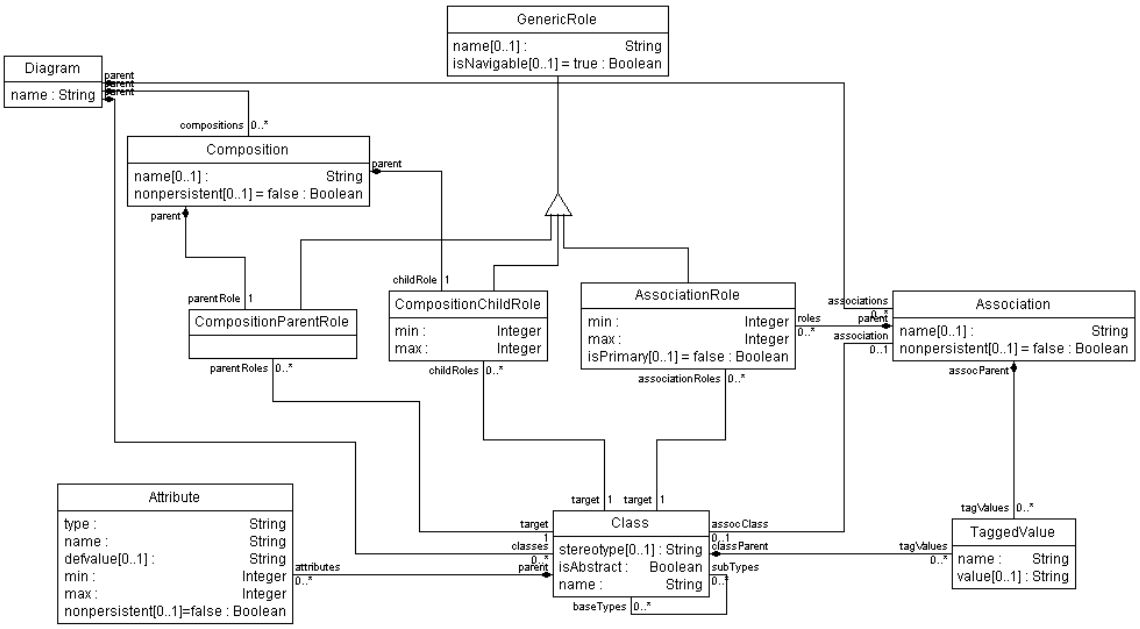

Figure 3 - The UML meta-meta model

The class definitions allow the creation of object instances. Such instances are not true objects, just handles (references) to objects that reside in the backend (similarly to the handle-body idiom defined by Coplien [7].). This has the following consequences:

1. An un-initialized instance variable is an empty reference (reference to *Udm::Null*).
2. Several variables may refer to the same instance, and simple assignment of variables does not imply the creation of a new instance.
3. New objects are always created by the *ClassName::Create()* static member function, which is automatically defined for all classes. This function expects the specification of a parent object (except for a single root object, every object must have a parent), and an optional child role.

Inheritance in the UML diagram is implemented as C++ public inheritance. Consequently, all attribute, composition, and association access methods of the base class are available in the derived classes. Multiple inheritance is also supported, with C++ inheritance relations converted to virtual as necessary.

All classes in the hierarchy are descendants of *Udm::Object*, which defines the generic functionality of objects in UDM. An object can decide its real type through the static variable *meta*. Each class has a static and constant type description object accessible via *classname::meta*. These allow the determination of compatibility between classes, objects and variables. The type information objects also provide reflection information (associations, attributes).

UML composition (containment) relationships are translated into access methods at both the 'child' and 'parent' side. These access methods return instances of wrapper classes that can be used to read and assign new values to the relationships. Objects can access their parent and children in two ways:

1. Access via composition relationship: returns objects only if they are linked together with the composition specified.
2. Access via parent/child type: returns all parent/child objects that match the specified data type (either directly or through inheritance), regardless of the composition relationship used.

A basic concept of UDM data networks is that objects are organized in a tree with a single root object, and thus all objects have a containing parent. In general, the lifetime of UDM objects is bound to their containment in the object tree. New objects are always created with their parent explicitly specified, and likewise, an object is deleted when no other object contains it any longer.

For each *UML attribute* an access method is defined in the corresponding C++ class. These access methods are named after the attribute name, and return an object. These objects can be converted into or assigned new values of a suitable data type. Supported UML data types are String, Integer, Boolean, and Real, which are mapped to the C++ data types string (as defined in C++ STL), integer, bool, and double respectively. In case of array attributes the object returned by the access method also supports the *[]* operator to return an object for a specific item in the array. This object can also be converted, modified or assigned new values of a suitable data type.

Associations are accessed in a way very similar to compositions, with the only difference that associations are symmetric. The access methods for both ends of the association are named after the corresponding association role names. Associations without names at either end are considered non-navigable in that direction, thus no corresponding access method is generated. The type of the wrapper object returned by an access method again depends on the cardinality of the corresponding end of the association: it can be read or written as a single variable (if the maximum cardinality is 1) or through an STL *set<T>* of compatible objects.

Subtypes, Instances and Archetypes are accessed through the *Derived(), Instances()* and *Archetype()* non-static methods, which are generated for each class. Derived and instantiated objects can be created with the *CreateDerived(), CreateInstance()* – also non-static, generated methods for each class.

As mentioned above, the UDM API only exposes smart pointers to the user. Thus traditional extension of the API via inheritance is not a suitable option. New attributes added to the inherited class will reside only in the pointer and not in the object. To overcome this problem, UDM allows the users to specify *volatile* attributes in the class diagram. These attributes reside only in memory and not in any of the persistent storage formats. Functionality however cannot be added in a similar fashion. For this reason the UDM implements the visitor pattern [11]. The code generator can generate (based on a command line switch) a base visitor class and the accept method for each generated class. Users interested in adding extra functionality can inherit from the base visitor and add new functionality and/or create their own traversal mechanisms.

UDM also provides (limited) support for metamodel evolution: the regenerated API can cope with existing models if either of the following conditions is met:

1. the new metamodel does not change or remove existing definitions(classes, containment/association relationships, attributes)
2. existing models which are to be accessed with the regenerated API do not contain instances of existing definitions which were removed or changed in the new metamodel.

Meta access and dynamic metamodel loading

During normal UDM operations, the objects of the metamodel (that represent the abstract syntax of the DSML) are instantiated in the generated .cpp file. The generated .h file provides a convenient API to access the objects representing the “terminal symbols” of the DSML.

As mentioned earlier, the UDM framework is capable of operating in a generic way, which enables access to objects made according to a specific DSML without having to compile in any generated code (that is specific code generated from the metamodels of the DSML, in our case UML class diagrams) in the application. This is achieved by loading the metamodel at run-time and then creating and accessing the DSML objects through a generic API. The domain-specific objects will have type pointers pointing to the (dynamically loaded) meta objects. The metamodel is dynamically loaded from the intermediate XML product between the interpreter and the code generator. Since that XML file also acts as the persistent form of a data network of objects of the “UML” DSML, it is loaded exactly the same way as any other model. This makes possible to develop highly generic applications, which can operate on objects created in various DSMLs relying completely on reflection through the metamodel loaded run-time. A good example for such an application is the UdmCopy tool, which is a binary executable that can create a copy (possibly in a different physical representation) of a data network created in any DSML. Even if the domains-specific API is missing, during generic access to the objects, and the C++ template classes are not instantiated, the generic API remains strictly typed and types are checked through reflection. Generic methods like *GetChildren(kind, role)* require meta-objects (*Uml::Class, Uml::AssociationRole*) as parameters instead of names.

The backends

XML Backend

The XML backend is based on an extended version of the standard XML/DOM interface, implemented by the Xerces-C++ parser [9]. The UDM generator tool - along with the API files- also generates the DTD for the XML files compatible with the interface definition. The basic XML-processing facilities provided are loading (parsing) an XML file into the backend, and saving the backend data structure into an XML file.

The XML backend imposes the following restrictions:

- XML does not have a good technique for representing associations and especially lacks support for association classes. These UDM features are thus implemented through specially named XML ID/IDREF attributes.

- The current version neither validates the data tree when it is modified, nor validates the tree when it is saved, so the validity of the generated XML document is not guaranteed. The recommended way to validate XML output is to reload it again as the last step of generation.
- There is a single root object, which is created as the first object when a data tree is opened (or loaded). The root object cannot be changed.
- The XML backend permits objects to exist without parents temporarily. If such an object is later attached to any parent, it will become a normal object and recorded in the persistent data.

GME Backend

The GME backend connects the interface to the MGA library[2], which is a component of the GME tool. The operation of the MGA library is also based on meta-information stored in its internal meta-database.

The GME backend provides functions for creating, opening, and closing GME databases, as well as an API for simple transaction control. It is also possible to create GME interpreters and other components based on UDM.

Since GME's metamodeling environment supports metamodeling constructs which are not supported in UML (atoms, models, FCOs, sets, members, references, aspects, connections, etc.[2]) a number of conventions have been defined on how to map these notions to UML notions:

- *GME atoms, models* are represented as *UML classes*
- *GME set-member, referenced-referee* relationships are represented as associations without an association class, with a special role names at their end points.
- *GME connections* can be represented as associations with or without association class, depending on whether the GME connection has attributes that need to be accessible from the UDM API.

The GME backend also provides transaction support on UDM data networks.

MEMory Backend

In the case of the MEM backend all the objects are stored in memory, in heavily hashed structures, which allow fast access to all associated objects (children and links). The hash keys are based on unique identifiers of the objects formed from their address in the memory – making "find-object" operations extremely fast.

The MEM backend also has a compact binary file-format, which can be used to save and restore static data networks. The MEM backend also can be used as a non-persistent backend, with the data network and all the objects created in runtime. As expected, in such cases there are no file operations at all. UDM uses a MEM data network (without persistence) for the meta-objects and meta-meta objects, which are created at startup time.

CORBA Backend

The CORBA backend provides support for marshaling/unmarshaling UDM data networks into a "flat", "network" form. It works on the data networks in the same manner as other backends, but it is slightly different from the other local backends: it uses remotely defined and stored metamodel information.

The key concept is (1) to have remotely defined metamodels maintained by a server application, such that the metamodels describe the same information as the local metamodels in the other cases, and (2) to have the client backend use these metamodels transparently, via a communication protocol, as if they were local metamodels. Thus, the programmer is able to work the same way with any UDM specific data using this remote metamodel, as if the data had a local metamodel.
The UDM API generator generates C++ class definitions giving access to all components of the remote metamodels. The CORBA backend was created and is being maintained in the OTIF (Open Tool Integration Framework) project [8].

The communication between the server and the clients using the CORBA backend relies on the CORBA mechanisms. A CORBA IDL file defines interfaces to access the (server-hosted) metamodels, as well the data structures used to transfer data objects between clients and the server. The CORBA backend requires one server application, which is responsible for maintaining and giving access to all the metamodel information. OTIF contains such a server that provides these services, and implements the interfaces specified in the same IDL.

The CORBA backend extends the core of the UDM framework: the *ObjectImpl* and *DataNetwork* pure virtual classes. The *CORBAObject* class extends the *ObjectImpl* class to support remote metamodel accessibility; however it does not support create and setter methods: the (remote) metamodel information is read-only. The *CORBADataNetwork* class extends the *DataNetwork* class with the functionality to send and receive UDM data networks to and from a server. The data networks are

created in a client, sent to the server, and fetched by a/another client from the server, using the CORBA backend. The send and receive operations transform (effectively: marshal and unmarshal) the data structures into a “network” format, and this process is hidden from the programmer. The CORBA backend was designed and implemented in that way that the programmer needs to do only minimal changes to the source code of an application when switching from using local to remote metamodels.

OCL Evaluator

Object Constraint Language (OCL) is a standard, predicate-logic oriented language that allows the user to describe sophisticated well-formedness and integrity rules over objects, which are very hard or impossible to express with only class diagrams.

The syntax and the semantics of OCL are close to other formal notations. A standard OCL expression consists of the context (Class) on which the rule will be evaluated and the equation that evaluates to a Boolean value regarding to whether the constraint is satisfied for a particular object or not. The language has features for retrieving attribute values, and for accessing association-ends specifying the role and/or the name of the class at the association-end. The user may traverse the entire data network via containments collecting the objects. OCL introduces predefined container types such as *Sets*, *Bags* and *Sequences* with which the user can iterate over the collections cumulating data (i.e. the generic *iterate* iterator) or may create more complex operations. OCL allows to simplify and to make a rule more readable using variable declarations or definitions (the latter are introduced by OCL 2.0). With these features it becomes possible to take a complex expression apart and to make the parts reusable in other constraints.

From the standard OCL rules (*invariants*, *preconditions* and *postconditions) o*nly invariants are applicable in UDM because the other ones are associated with methods that UDM does not support. The constraints as well as definitions (mentioned above) can be specified in GME UML as part of the class diagram.

In the application code, the user may access OCL related objects over the generated API (using *constraints()* and *definitions()* calls on *Udm::Class*). Since UDM allows alteration of the metamodel at run-time, the expression of a rule can be modified, even new definitions may be introduced or constraints can be eliminated.

To evaluate the constraints the user has to call the OCL engine with the *ocl::Initialize()* command. Since constraints found in the metamodel are parsed and analyzed during the initialization, it always has to be done at least once before evaluating and/or if the constraints were altered. The *ocl::Evaluator* class plays the main role in the evaluation process, and its constructor requires the context of the execution (i.e. a *udm::Object*) with an optional set of constraints obtained from classes. If this set is empty, all applicable rules will be evaluated for the object or for a sub-tree defined by the context. The constraint checking is performed by *ocl::Evaluator::Check()* method. The process is controlled with the optionally supplied *ocl::SEvaluationOptions* structure. The options include the usage of short-circuit operators and iterators, depth of the sub-tree to be traversed, or kind of the result (exception or false value in case of constraint violations). The user may specify when the evaluation must terminate (after the first violation, after all constraints are checked).

OCL related meta-objects are always available in the metamodel but the user is not required to evaluate or to parse them. The *UdmOcl* library can be optionally included and the core (parser and evaluator) of the module is so generic that it may be extended for other tools or frameworks.

Comparison with other tools

UDM provides a metamodel-based tool for building implementations for DSMLs. Domain specific code generators, data exporters, importers can then be written using the API that was generated from the metamodels by the UDM tool(s).

GME provides a COM API to access model and metamodel information. The API is specific to GME and not to the domain being developed. GME also has an export/import utility that exports to and imports from XML. The XML format is a proprietary format suitable to store any GME model database [2].

In MetaEDIT+ version 4.0 an API has been added to access the models. The API uses SOAP and it is domain independent. MetaEDIT+ also has an export/import facility using XML. The XML format is again specific to MetaEDIT+ and not the domain [3].

Based on our experience, programmer productivity is much lower using a generic API, than a domain-specific one. The reason being, many errors will be detected much later in the development. For example to retrieve an object in GME the function is *GetModels(“kindName”)* and in MetaEDIT+ it is *findObject(“kindName”,”name”)*. If the kind name of the model is incorrect, it will be detected at run-

time and not at compile time. UDM, on the other hand, generates a specific API and a specific XML format for each domain. This helps increase productivity by providing a clean interface and eliminating many potential errors. The specific XML format can be used to exchange domain specific data with other tools that do not need to know an XML format of a particular UDM-based tool.

At the implementation level UDM is a data binding tool and can be compared with other data binding tools.

ObjectSpace Inc. developed an XML based tool called DXML, which is part of Recursion Software's Voyager® Application Server product. The tool uses a DTD to generate a Java based API. The API wraps the xml files and allows users to create and modify XML tags by creating and modifying java objects [12].

Zeus is an open source initiative by Enhydra.org. It generates and provides a Java API to access and modify XML files. Given a DTD, a Java interface and implementation is generated. Marshalling and unmarshaling code is also generated that can convert XML files to a java object representation and back [14].

Castor by Exolab Group provides a generic java object to XML marshalling and unmarshaling framework. The framework allows "the marshalling of any bean-like java object to and from xml" [15]. Castor can also generate the Java classes from XML schemas [17].

The .NET framework by Microsoft has an XML data-binding tool called *XSD*. XSD can generate C#, Visual Basic .NET and JScript.NET API for a given XML schema. XSD can also generate XML schema files from any .NET source file. The API and XML files are bound using a generic tool called XmlSerializer [18].

The Eclipse Modeling Framework (EMF) is an open source tool developed using the Eclipse framework. Given an XMI file representing the data model it can generate a Java based API that implements the class diagram specified in XMI. The objects of the generated API can then be serialized to an XML representation. The generated API can also be extended [16].

All the tools mentioned above can create an API from either a DTD or some XML file. Some tools such as Castor and .NET can also create XML Schemas from existing code. UDM and EMF on the other hand create the DTD and code API from class diagrams represented using XMI[6]. The API generated by DXML is not always intuitive and lacks type safety [13] while Zeus, Castor, .NET and EMF require marshaling/unmarshaling of the data. The UDM API allows users to directly manipulate the XML files. The UDM framework supports multiple files formats such as XML, MGA [2] and MEM. It also supports XMI: UDM metadata can be exported in XMI format. Furthermore, UDM has a generic API layer that allows users to discover the structure of UDM based objects and then manipulate them at run-time [18].

Application experience

The UDM framework has been used as the implementation environment for various model transformation and code generation tools that operate on models. There are currently several projects using the UDM framework. The two biggest projects are the OTIF (Open Tool Integration Framework) [8]and the GReAT (Graph Rewriting and Transformation) Code Generator [10] project. In the OTIF project we extended the basic UDM framework with remote metamodels based using CORBA. GreAT employs UDM to produce a C++-based translator from a graph-transformation based, high-level representation of the translator. The generated translator, in turn, employs UDM to access, traverse, and manipulate models.

In general, the code generated by UDM features:

- *Robustness:* The strictly typed UDM interface forces early, compile-time errors, while the run-time type checking facility built into UDM eases the validation of data networks.
- *Efficiency*: The proxy pattern used in UDM, by controlling access to a real object, defers the full cost of the object's creation and modification until the object is actually modified.
- *Traceability*: The UML notions (inheritance and composition) are reflected in OO fashion in the generated C++ interface. Also, because the UDM access methods have the same name as the attributes and associations in the model, the generated code is easy to trace or debug.
- *Flexibility*: Platform independence and multiple backends support a wide variety of applications.

We gained the following benefits of the UDM infrastructure:

- *Rapid implementation process*: The programmer can design the application with a graphical interface (GME) and get a ready-to-compile C++ code corresponding to the designed UML diagram without implementing a single line of source code.
- *Highly customizable solution*: Any modification of the designed UML diagram propagates trivially to the source code.

- *Reduced development costs*: The speed of the development and the highly customizable nature of the framework allow reduction in the development effort.

Summary, conclusions

We have described UDM: a framework for implementing a data layer in support of a DSML. UDM is generative techniques: the implementation is generated from UML class diagrams used as metamodels, it provides a number of backends for the physical implementation and persistence of data structures, it has a constraint evaluator that checks integrity constraints, it supports reflection, and it has been successfully used in implementing a number (~10) of model transformation tools [10]. Obviously, there is a performance penalty when used, but in our experience the price for flexibility and generality was acceptable. UDM is a part of our infrastructure software and we plan to develop it further.

Acknowledgement

The DARPA/IXO MOBIES program and USAF/AFRL has supported under contract F30602-00-1-0580, in part, the activities described in this paper. The first implementation of UDM was created by Miklos Maroti.

References

[1] J. Bézivin, “Tooling the MDA framework: a new software maintenance and evolution scheme proposal”.
[2] The Generic Modeling Environment (GME 2000), http://www.isis.vanderbilt.edu/projects/gme/Doc.html
[3] Steven Kelly, “Improving the Integration of a Domain-Specific Modeling Tool”, Workshop on Tool Integration in System Development, ESEC/FSE, Helsinki, Finland, 2003.
[4] Tony Clark, Andy Evans, Stuart Kent: Engineering Modelling Languages: A Precise Metamodelling Approach. FASE 2002: 159-173
[5] OMG Unified Modeling Language Specification, Version 1.4 draft, February 2001, http://www.omg.org
[6] OMG XML Metadata Interchange (XMI) Specification, Version 1.2, January 2002, http://www.omg.org
[7] J.O. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992
[8] OTIF (Open Tool Integration Framework), http://micc.isis.vanderbilt.edu:8080/Portal and http://www.isis.vanderbilt.edu/Projects/WOTIF/default.html
[9] The Xerces C++ Parser, http://xml.apache.org/xerces-c/index.html
[10] http://www.isis.vanderbilt.edu/Projects/mobies/default.html.
[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns”, Addison-Wesley, 1995.
[12] Recursion Software Inc., Voyager® Application Server 4.6 Data Sheet, http://www.recursionsw.com/products/voyager/datasheets/appserver_ds.asp
[13] DXML Review, http://www.jfind.com/listings/74.shtml, Jfind.com.
[14] Zeus, http://zeus.enhydra.org/index.html, Enhydra.org.
[15] The Castor Project, http://castor.exolab.org/index.html, Exolab Group.
[16] “The Eclipse Modeling Framework (EMF) Overview”, Sept. 8, 2002.
[17] Ronald Bourret, “XML Data Binding Resources”, http://www.rpbourret.com/xml/XMLDataBinding.htm, May 2, 2003.
[18] Niel Bornstein, “XML Data-Binding: Comparing Castor to .NET”, http://www.xml.com/pub/a/2002/07/24/databinding.html, xml.com, July 24, 2002.