VANDERBILT UNIVERSITY

INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37235

Evolving Paradigms and Models in Multi-Paradigm Modeling

Daniel Balasubramanian, Chris vanBuskirk, Gabor Karsai,
Anantha Narayanan, Sandeep Neema, Ben Ness, Feng Shi

TECHNICAL REPORT

ISIS-08-912

Evolving Paradigms and Models in Multi-Paradigm Modeling

Daniel Balasubramanian Chris vanBuskirk Gabor Karsai

Anantha Narayanan Sandeep Neema Ben Ness Feng Shi

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, 37203

Abstract

The essence of model based software development for domain specific applications is in the definition of a meta-model that captures the key aspects of the domain. However, the meta-model rarely defines the domain completely, and must often be modified to reflect our improved understanding of the domain during the course of development and subsequent use of domain specific software. The modification of the meta-model creates a versioning problem, whereby existing models may no longer conform to the modified meta-model. This report describes the Universal Model Migrator, a tool that allows domain designers to declaratively specify incremental modifications to their meta-models, in order to facilitate the automatic evolution of domain models to remain conformant to the latest version of the meta-model.

1 Introduction

The use of model based software development techniques has expanded to a degree where it may now be applied to the development large heterogeneous systems. Due to their high complexity, it often becomes necessary to work with a number of different modeling paradigms in conjunction. Model based development tools, to a large extent, are living up to this challenge. However, short turnover times mean that only a limited time may be spent in defining meta-models for these modeling paradigms before users begin creating domain specific models. Deficiencies, inconsistencies and errors are often identified in the meta-models after development is well underway, and a large number of domain models have already been created. Changes may also result from our improved understanding of the domain over time, and other modifications in the domain itself. Newer versions of meta-models must therefore be created, which may no longer be compatible with the large number of existing models. The existing models must then be recreated or manually evolved using primitive methods, adding a significant cost to the development process. The problem is especially acute in the case of multi-paradigm approaches, where multiple modeling languages are used and evolved, often concurrently.

Existing solutions to this problem are primitive, ad-hoc techniques, that often resort to directly specifying the alterations in terms of the storage format of the models. One such approach is the use of XSL transformations to evolve models stored in XML. Database schema migration techniques have been applied to migrate models stored as relational data. These approaches are often nothing more than pattern based replacement of specific character strings, and do not capture the philosophy driving a meta-model change. When dealing with complex meta-models covering multiple paradigms, comprehension is quickly lost when capturing meta-model changes using these methods.

Automatic evolution of domain models to conform to changes in meta-models is clearly an important requirement. The responsibility to satisfy this requirement falls on the tool vendors who enable model based development technology. However, it is a challenge to provide a generic solution that may be applicable in the variety of scenarios where domain specific modeling tools may be used. We need a machinery that is generic in the sense that it may be applicable to any meta-model irrespective of the domain, and also be comprehensible to the domain designers with relevance to their specific domains. In this paper, we describe a highly generic approach to migrating domain models based on changes in the meta-models. We present a suite of tools, using which the migration can be specified by clearly capturing the change in the meta-model, and the action necessary to migrate the domain models. We believe that approaches like this are essential for the support of engineering processes that use multiple modeling paradigms that need to evolve over the lifetime of the system.

2 Background and Rationale

In this paper we assume that modeling paradigms are defined using a metamodel. A metamodel M_L defines a modeling language L by defining its abstract syntax, concrete syntax, well-formedness rules, and dynamic semantics (which is usually defined by a mapping from the abstract syntax into a semantic domain) [7]. Here, we are focusing on the abstract syntax of the modeling paradigm.

The abstract syntax essentially defines the data model for the constructs of the modeling language, i.e. the data structures that could hold the models, as physical data, in a form independent of the concrete syntax (which could be textual or graphical). There are various techniques for specifying the abstract syntax for modeling languages, the most widely used is the Meta-object Facility (MOF) [11], but for clarity here we will use UML class diagrams that are visual and often more flexible than MOF. Hence, we assume that the metamodel of the modeling language is defined by and is expressed as a UML class diagram. The examples in this paper use UML class diagrams with stereotypes indicative of the role of the element, such as *Model*

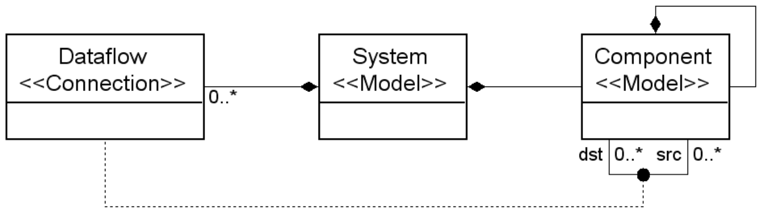


Figure 1: Example meta-model

or *Connection* - but they may be understood as simple UML classes. Note that the actual models are object diagrams that are compliant with the UML class diagram of the metamodel.

2.1 Rationale

It has been our experience that when a modeling language evolves over time, most evolution steps are incremental changes that affect small portions of the language locally. For instance, consider a meta-model for the domain of Dataflow diagrams. Figure 1 shows a meta-model for this domain. According to this meta-model, a Dataflow "System" is a model that contains one or more "Components". Components may contain other Components. Components may be connected to each other via "DataFlow" connections.

A possible and likely evolution of this language is that the domain designer may want the DataFlow connections to be routed through Ports within the Components. This change is effected in the meta-model by defining new types called "inPort" and "outPort" to be contained within Components, as shown in Figure 2. The DataFlow connection is modified to connect Ports instead of Components. The models conforming to the first version of the meta-model, with DataFlow connections between Components, will no longer be valid in the new version of the meta-model. For this domain, the steps necessary to migrate these old models to conform to the new meta-model can be stated as: for every Dataflow connection, create a Port within the Component at each end, and connect the Dataflow connection between these Ports.

Another possible change in the language could be that the domain designer no longer wants Components to be composed hierarchically. She may therefore remove the containment relation between Components. In this case, to make the old models conformant to the new meta-model, Component hierarchies must be "unwound" throughout the model, requiring a deep traversal with recursive treatment.

In our experience, the latter kind of evolution is rare in DSMLs, while evolutions of the former category are more frequent. Our MCL has been designed to easily specify changes of the former nature, by addressing changes to specific model elements

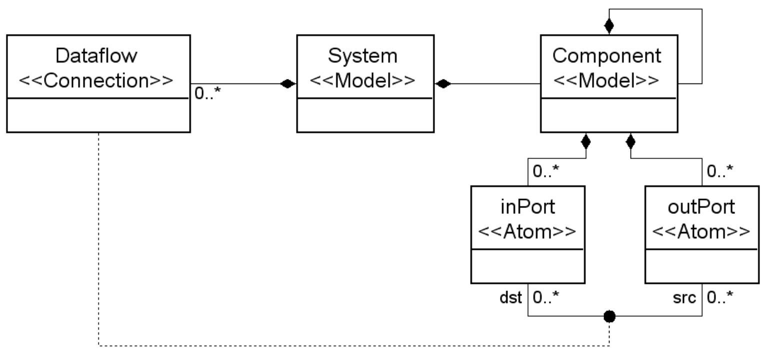


Figure 2: Evolved example meta-model

locally.

2.2 Hypothesis

Our essential hypothesis is that evolutionary changes on the modeling language will be reflected as changes on the metamodel, i.e. the UML class diagram. When the modeling language is evolved the language designer has to modify the metamodel, and the new version of the language will be compliant with the new metamodel. The key observation here is that metamodel changes are explicit, and these changes could be used to automatically derive the algorithm to migrate the models of the old modeling paradigm to the models compatible with the new paradigm. Here, we make an essential assumption: changes performed on the metamodel are known and well-defined, and all these changes could be expressed in an appropriate language. We call this language the Model-Change Language (or 'MCL').

A key design decision for MCL is that it is used to represent only the changes performed on the metamodel. During modeling language evolution, changes are incremental, and most of the metamodel elements are unaffected. Representing only the incremental changes the complexity of specifications in MCL is much reduced. We assume that whatever is not represented in MCL remained the same from one metamodel version to the next.

The need for a new language to describe changes is not immediately obvious when the 'old' and 'new' metamodels are already available. The answer is that computing the changes from the existing metamodels could be exceedingly difficult and it is unclear if the problem is solvable in a fully automatic, algorithmic way. An algorithm would have to perform a sophisticated differencing between the two metamodels and find out exactly where the differences are, and then infer how 'old' modeling concepts (classes, associations, attributes, etc.) map into 'new' modeling concepts, i.e. the intent of the metamodel designer would have to be recovered. While this is a challenging research problem, it is unclear whether such inference could be made fully automatic. For instance, much semantic information is carried in the names of classes, and an automated tool would have a problem with figuring out that 'Failure Mode'-s in an 'old' metamodel are to be mapped into 'Fault Mode'-s in the 'new' metamodel. Here, we took the less sophisticated yet pragmatic method of asking the designer to provide a specification of the changes in MCL.

Finding the right level of abstraction for MCL was a challenge. MCL could have been very low-level, and describe changes using simple editing operations (delete/add classes, associations, attributes) but specifying such level of details would have been cumbersome. Finally, we opted for an idiomatic approach, where specific, well-understood metamodel changes were identified as idioms (i.e. reusable patterns of change operations), for which we developed a graphical notation. The important feature of the idioms is their compositionality: more complex metamodel changes could be compiled by combining idioms. This principle is similar to the approach used for metamodeling in the metaprogrammable GME tool [9].

In the subsequent sections we first introduce MCL, then describe the implementation of the language, compare it to other, similar efforts, and then summarize the results and discuss future research topics.

3 The Model Change Language

The basic pattern that describes a meta-model change, and the required model evolution, consists of an LHS element from the old meta-model, an RHS element from the new meta-model, and a "MapsTo" relation between them, as shown in Figure 3. For the sake of flexibility, it is possible to specify additional mapping conditions or imperative commands along with the mapping. This basic pattern is extended based on various evolution criteria, as explained below.

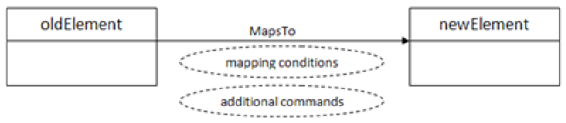


Figure 3: *MapsTo* relation to specify mapping of classes

The changes made to a meta-model will largely fall under one of the following categories described below.

3.1 Adding elements

A meta-model may be extended by adding a new concept into the language, such as a new Class, a new Association, or a new Attribute. In most cases, old models

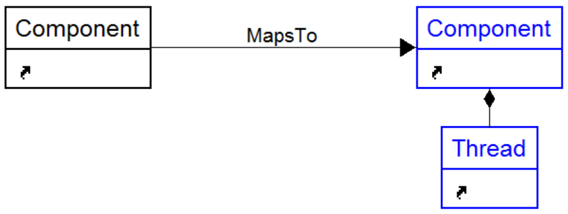


Figure 4: MCL rule for adding a new element

are not affected by the new addition, and will continue to be conformant to the new language, except in certain cases.

If the newly added element represents some model information within a different element in the old version of the meta-model, the information must be appropriately preserved in the instance models. In fact, this falls under the category of "modification" of representation, and is described further below.

If the newly added element plays a role in the well-formedness requirements, then the old models will no longer be well formed. The migration language must allow the migration of such models to make them well formed in the new meta-model. For instance, suppose that the domain designer adds a new element called "Thread" within a Component - and adds a constraint that every Component must contain at least one Thread. The old models can then be migrated by creating a new Thread within each Component, as shown in Figure 4.

3.2 Deleting elements

Another change to a meta-model may be the removal of an element. If a type is removed, and replaced by a different type, it implies a modification in the representation of existing information, and is handled further below. On certain occasions, elements may be removed completely, if that information is no longer relevant in the domain. In this case, their representations in the instance models must be removed. The removal of an element is specified by using a "NULL-Class" symbol in MCL, as shown in Figure 5.

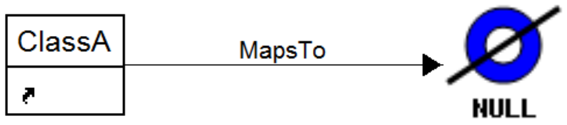


Figure 5: MCL rule for deleting an element

This implies that all instances of ClassA in the model are to be removed. Removal of an object may result in the loss of some other associations or contained objects. MCL follows the policy that in such cases, the loss of information is an error, unless these associations and classes were also explicitly marked to be deleted. If an unmarked object is lost, the user is notified of the error.

3.3 Modifying elements

The most common change to a meta-model is the modification of certain entities, such as the names of classes or their attributes. The basic "MapsTo" relation shown in Figure 3 suffices to specify this change. The mapping of related objects is not affect by this rule. If other related items have also changed in the meta-model, their migration must be specified using additional rules.

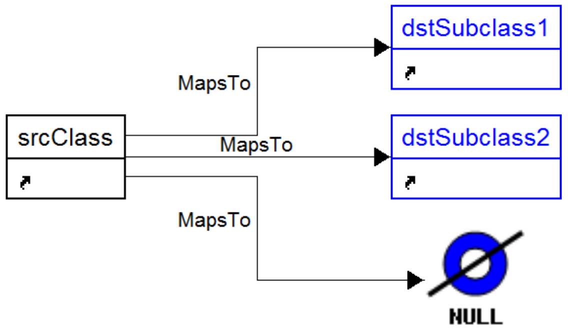


Figure 6: MCL rule for subclasses

Another type of modification in the meta-model is adding new sub-types to a class. In this case, we may want to migrate the class' instances to an instance of one of its sub-types. Figure 6 shows an MCL rule that specifies this migration. The subtype to be instantiated may depend on certain conditions, such as the value of certain attributes in the instance (this is encoded within the migration rule using a boolean condition for each possible mapping). The rule in Figure 6 states that an instance of srcClass in the original model is replaced by an instance of dstSubclass1 or dstSubclass2 in the migrated model, or deleted altogether.

As in the earlier case, other related objects are unaffected by this rule. If a related object violates a condition in the new meta-model, or is lost due to deletion, other MCL rules must handle these entities appropriately. Otherwise, the user is notified of the error.

3.4 Local structural modifications

Some more complex evolution cases occur when changes in the meta-model require a change in the structure of the old models to make them conformant to the new meta-model. Consider a meta-model with a three level containment hierarchy, with a type

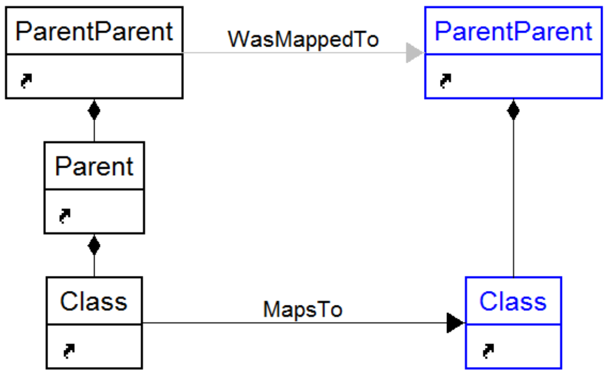


Figure 7: MCL rule for changing a containment hierarchy

Class contained in *Parent*, and *Parent* contained in *ParentParent*. Suppose that this meta-model is changed by moving *Class* to be directly contained under ParentParent. The intent of the migration may be to move all instances of *Class* up the hierarchy. The MCL rule to accomplish this is shown in Figure 7 (the *WasMappedTo* link is used to identify a previously mapped parent instance).

Note that this rule only affects *Class* instances. The other entities remain as they are in the model. Any *Parent* instances within *ParentParent* remain unaffected. If *Class* contained other entities, they continue to remain within *Class*, unless modified by other MCL rules.

3.5 Idioms and Complex Rules

Based on the descriptions given above, we created a set of idioms that capture the most commonly encountered migration cases. Some of the idioms were shown in Figures 4, 5, 6 and 7. We earlier described a case where associations (connections) may be rerouted using “ports”, by creating a “port” for each connection end within the models. Figure 8 shows the idiom for rerouting associations. The specific case shown here is rerouting associations through ‘ports’ that are contained model elements under some container. In the old language we had ‘inAssociationClass’-es between ‘inSrcModel’-s and ‘inDstModel’-s, and the new language the same association is present between the ‘Port’-s of the ‘outSrcModel’ and ‘outDstModel’ classes that were derived from the corresponding classes in the old model. The “WasMappedTo” link is used to find the node corresponding to the old association end. For the correct results, the new association ends must be created before the “MapsTo” can be processed for the association, and this is enforced by the use of the “WasMappedTo” link.

The MCL also provides primitives to specify the migration of attributes of classes in the metamodel. Attributes may be mapped just like classes, and the mapping can

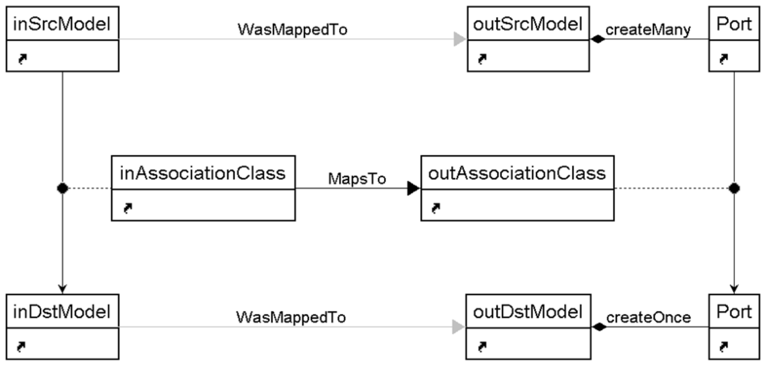


Figure 8: MCL rule for rerouting associations

perform type conversions or other operations to obtain the new value of the attribute in the migrated model. Figure 9 shows an MCL for migrating attributes.

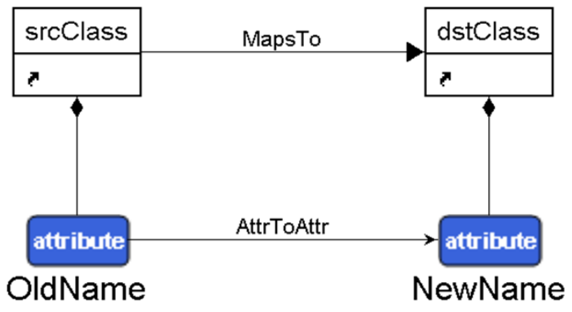
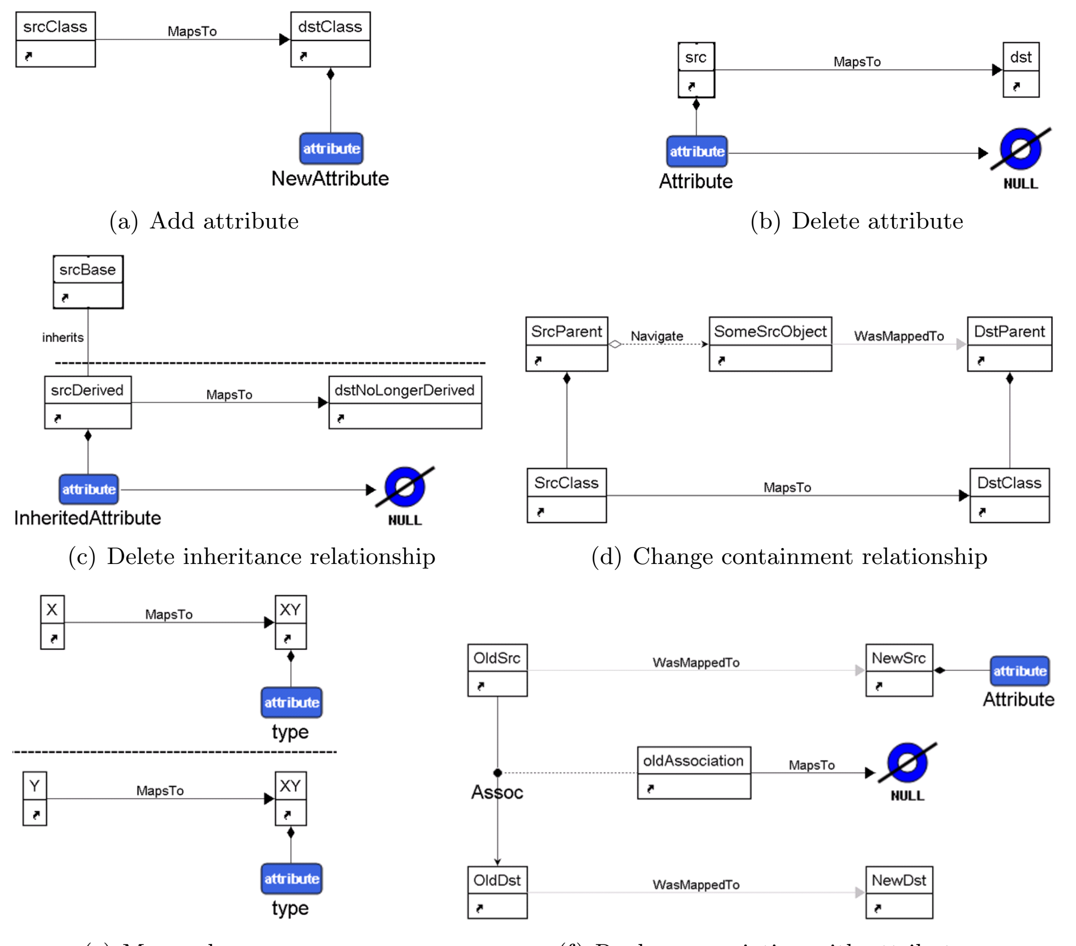


Figure 9: MCL rule for migrating attributes

In addition to the idioms listed so far, the tool suite for model migration will provide a library of similar idioms to handle common migration cases. Some of these idioms are shown in Figure 10 and summarized here. Figure 10(a) shows the idiom for adding a new attribute to some class in the metamodel. If the newly added attribute is mandatory, then it must be set in old models that did not have the attribute. A default value can be added for the attribute in the idiom, or a function may be added to calculate a value for the new attribute based on the values of other attributes in the instances. Figure 10(b) shows the idiom for deleting an attribute. This is similar tot he case of deleting classes. Figure 10(c) shows an idiom for the case when an inheritance relationship has been removed from the metamodel (the portion above the dashed line is not part of the rule, but shown for clarity). If the derived class had an inherited attribute, this will no longer be present in the migrated model, and must therefore be deleted.

Figure 10(d) shows an idiom for changing a containment relationship in the metamodel. This is a variation of the idiom shown earlier in Figure 7, for a more generic



(a) Add attribute

(b) Delete attribute

(c) Delete inheritance relationship

(d) Change containment relationship

(e) Merge classes

(f) Replace association with attribute

Figure 10: Idioms for model migration

case. This idiom also introduce a generic primitive called “Navigate”. It can be used to locate objects in the instance model by following a “navigation condition” starting from the object on the left end of the “Navigate” link, and use this object to determine the new parent in the migrated model. Figure 10(e) shows an idiom for merging two classes in the metamodel into a single class, possibly adding an attribute to record its old type. This is effected using two migration rules, separated by the dashed line. The migration rule can encode a command that will set the value of the attribute based on its original type. Figure 10(f) shows an idiom for the case where an association in the metamodel is replaced by an attribute on the source side of the metamodel. This is accomplished by mapping the association to a “null” class (similar to the ‘delete class’ case), and adding a new attribute on the source side.

These idioms may also be composed together to accomplish more complex evolutions. A complex migration case may be handled by putting together different relevant idioms. Figure 11 shows an example where different idioms have been com-

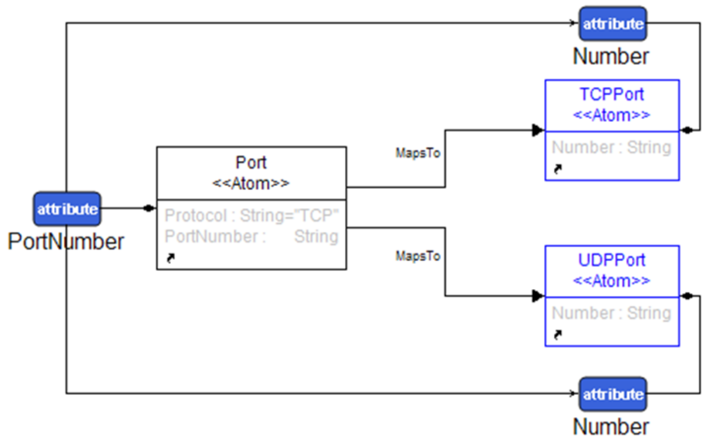


Figure 11: A complex MCL rule

bined into a complex migration rule. In this rule, *Port* can be mapped to either a *TCPPort* or a *UDPPort*. The condition for the mapping is decided by the string attribute “Protocol”. The “Number” attributes of the new classes is determined from the “PortNumber” attribute of the old class.

4 The Implementation of MCL

We have implemented MCL in the context of two existing tools for domain-specific languages: the metaprogrammable Generic Modeling Environment (GME) [8], and the Universal Data Model (UDM) [10]. GME is a visual tool for creating both metamodels and domain (instance) models, and UDM is a meta-programmable data layer package that provides programmatic (C++) access to models, both for model creation and model transformation. Our overall approach is the following.

1. The language engineer defines the first version of the metamodel. As mentioned above, this metamodel is an UML Class Diagram.
2. Domain-specific models that conform to the metamodel defined above are created. These domain specific models are usually created either with interactively or programmatically.
3. When the metamodel needs to be changed, it is modified, resulting in a new version of the metamodel, which is another UML Class Diagram.
4. In order to upgrade the existing models so that they conform to the new version of the metamodel, the language engineer creates an MCL model describing the changes between the old and new metamodels. From this model, executable

code is generated that implements the transformation from models conforming to the old metamodel to models that conform to the new version of the metamodel.

For the last step above, MCL has been implemented as a domain specific language with a metamodel of its own. An MCL model contains (1) the metamodels for the 'old' and the 'new' versions of the modeling language as UML class diagrams, and (2) model migration rules based on the idioms described in the previous sections. The migration rules are visually defined by using references to the metamodel elements (UML classes) of the old and new metamodels. Each transformation rule describes the mapping between one or more elements in the old metamodel to element(s) in the new metamodel. Using these transformation rules, a code generator produces executable code that takes as input a model conforming to the old version of the metamodel, and produces a model that conforms to the new version of the metamodel. The structure of this code generator is described below.

4.1 The Code Generator

The code generator produces the executable transformation program that traverses the input ('old') models and incrementally builds up the target ('new') models. We will first describe how the generated code should transform an input model, which will give the motivation for our code generator's algorithm.

Assuming that we know how to "map" all types of elements in 'old' models to elements in the 'new' models, we can use the following simple algorithm to facilitate the transformation. Model databases always have one singleton "root" model object, so we can start with this object, examine its type, and create the corresponding 'new' model object that conforms to the new version of the paradigm, assuming we know how to map elements of every type. We then continue this mapping in a depth-first manner, recursively mapping the contained "child" elements of the current object. Finally, after all elements in the old model are "migrated" into the appropriate elements in the new model, associations (i.e., connections between elements) in the old model are migrated into associations or to the appropriate elements in the new model, in a second, depth-first pass over the input model. This summarizes the high-level algorithm used by the generated code to migrate models.

We assumed above that as individual elements in the old model are traversed, the generated code has knowledge about the corresponding element(s) to be created in the new model. In terms of code, this can be implemented by having a function for every type of element in the old metamodel. Then, as the elements in the old model are traversed in a depth-first manner, their type is examined, and they are passed to the appropriate function which then creates the corresponding concept(s) in the new model. Thus, the logic of the migration needs to go into these individual functions. The pseudocode of the code generator algorithm is given in Table 1.

Because the data access layer package (UDM) offers a reflective API, we can

```
// Code generator algorithm for model migration code
for all elements x in the old metamodel
create a function f to migrate x
  if the user has defined mappings for x in the MCL model
  then use the user defined mapping in f
    else
      look in the new metamodel to see if there is an element y
        with the same name as x
      if such an element y exists then x is mapped to y in the generated code
      else x is not mapped, signal an error about potential loss of information
  add code to f that retrieves all children of x and passes each
    to the appropriate migration function
```

Table 1: Code generator algorithm

easily query the metamodels (contained in the MCL model) for information such as the types of children an individual object can have.

The generated migration functions for each element X in the old metamodel take two parameters: an instance of X, which is the element in the input model that the function is migrating, and a reference to the parent object in the output model in which the corresponding element for X will be created. The function performs the migration of an element of type X according to the code that was generated (by the algorithm above) from the migration rules defined by the user, then finds the children elements of X, and for each one, calls the appropriate migration function, passing in the current child element of the X instance (the next element to be migrated) and the object to which the current X instance was mapped in the new model (this will be the parent of objects created in the migration function being called). This process continues in a depth first manner until all objects in the old model have been traversed.

5 Related Work

Our work presented here has its origins in the technology for database schema evolution in object-oriented databases [4]. However, models and modeling languages are typically richer semantically than object-oriented data, hence the increased complexity in migration operations.

For model-based tools, the prevailing current approaches to paradigm evolution are either manual reconstruction, simple rewriting scripts, or general purpose graph-rewriting tools like [1] [2] [12]. Note that all of these are general purpose transformation approaches, and not domain-specific. Data and program transformation, as approached in [3] [5] exhibits a similar problem. However, program transformation for the purpose of optimization does not address the problem of rewriting the input into another language.

The work presented here is highly relevant to process-centered model engineering [5], where the modeling languages themselves evolve. Our previous work [13] developed the origins of the approach described here, which has resulted in an automatically generated XSLT translation script. The work presented here extends the results with a much richer set of migration operators, as well as a code generator that produces executable code in a procedural language to execute model migration.

6 Summary and Future Work

We have presented an approach to the automated evolution of models when their metamodel changes. The approach is based on the explicit modeling of how the metamodel have changed, and on the automatic generation of executable transformation code from these models of the changes. The work has identified a number of useful idioms for model migration, and showed how these could be composed to form complex migration operations. A language, called MCL, has been developed for representing the changes, and supporting tools have been created that synthesize the executable code from these declarative specifications.

The model migration problem is an essential one for model-driven development and tooling, and there are several challenging problems remaining in this area. One, highly critical issue is that of semantics: how can we ensure that the migrated models do preserve semantics? It is very hard to answer this question in general, but for specific cases one could envision tools that validate that semantics is preserved in some sense - which could be, arguably, defined using an extended MCL. Efficiency of the migration code is also of paramount importance, especially on large-scale models. The migration idioms that we have developed were based on our past experience, but it seems that this should be an evolving set, to be extended and refined by other developers. Thus, a continuation of this work would need to address the problem of supporting such an extensible migration idiom set. Finally, model migration is only part of the problem - we often need to migrate the tools that process those models. Hence, further research is needed on automatically migrating the executable code of the tools based on the metamodel changes.

7 Acknowledgments

The work described in this paper has been supported by the Defense Advanced Research Projects Agency, the National Science Foundation, and the US Army Research Office. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above agencies or the U.S. Government.

References

[1] Aditya Agrawal, Tihamer Levendovszky, Jonathan Sprinkle, Feng Shi, and Gabor Karsai. Generative programming via graph transformations in the model-driven architecture. In OOPSLA, 2002: Workshop on Generative Techniques in the Context of Model Driven Architecture, November 2002.

[2] D. H. Akehurst. Model Translation: A UML-based specification technique and active implementation approach. PhD thesis, University of Kent at Canterbury, United Kingdom, December 2000.

[3] Robert Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In Proceedings of the 2nd International Conference on Software Engineering, pages 337-344, 1976.

[4] Banerjee, J., Kim, W., Kim, H., and Korth, H. F. 1987. Semantics and implementation of schema evolution in object-oriented databases. SIGMOD Rec. 16, 3 (Dec. 1987), 311-322. DOI= http://doi.acm.org/10.1145/38714.38748

[5] Erwan Breton, Jean Bezivin, "Process-Centered Model Engineering," edoc, p. 0179, Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001

[6] S. Gerhart. Correctness-preserving program transformations. In Proceedings of the 2nd ACM SIGACT-SIGPLAN

[7] Karsai, G.; Sztipanovits, J.; Ledeczi, A.; Bapty, T.; Model-integrated development of embedded software, Proceedings of the IEEE, Volume: 91, Issue: 1, Jan. 2003 Pages:145 - 164

[8] Ledeczi, A.; Bakay, A.; Maroti, M.; Volgyesi, P.; Nordstrom, G.; Sprinkle, J.; Karsai, G.: Composing domain-specific design environments, IEEE Computer, Nov. 2001, Page(s): 44 -51.

[9] Ledeczi A.; Maroti M.; Bakay A.; Karsai G.; Garrett J.; Thomason IV C.; Nordstrom G.; Sprinkle J.; Volgyesi P.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, 2001.

[10] Magyari E.; Bakay A.; Lang A.; Paka T.; Vizhanyo A.; Agrawal A.; Karsai, G.: Udm: An infrastructure for implementing domain-specific modeling languages, The 3rd OOPSLA Workshop for Domain-Specific Modeling, October 2003.

[11] Meta-Object Facility – Standards available from Object Management Group, www.omg.org

[12] Schürr, A. Programmed Graph Replacement Systems, pages 479-546. World Scientific, Singapore, 1997.

[13] Sprinkle J., Karsai G.: A Domain-Specific Visual Language for Domain Model Evolution, Journal of Visual Languages and Computing, 15, 2, April, 2004.