Continuous Migration Support for Domain-Specific Languages

Daniel Balasubramanian, Tihamer Levendovszky,
Anantha Narayanan and Gabor Karsai
Institute for Software Integrated Systems
2015 Terrace Place
Nashville, TN 37203
{daniel,tihamer,ananth,gabor}@isis.vanderbilt.edu

ABSTRACT

Metamodel evolution is becoming an inevitable part of software projects that use domain-specific modeling. Domain-specific modeling languages (DSMLs) evolve more frequently than traditional programming languages, resulting in a large number of invalid instance models that are no longer compliant with the metamodel. The key to addressing this problem is to provide a solution that focuses on the specification of typical metamodel changes and automatically deduces the corresponding instance model migration. Additionally, a solution must be usable by domain experts not familiar with low level programming issues. This paper presents the Model Change Language (MCL), a language and supporting framework aimed at fulfilling these requirements.

1. INTRODUCTION

Model based software engineering has been especially successful in specific application domains, such as automotive software and mobile phones, where software could be constructed, possibly generated from models. A crucial reason for this has been the tool support available for easily defining and using domain specific modeling languages (DSMLs). However, the quick turnover times required by such applications can force development to begin before the metamodel is complete. Additionally, the metamodel often undergoes changes when development is well underway and several instance models have already been created. When a metamodel changes in this way, it is said to have *evolved*. Without supporting tools to handle metamodel evolution, existing instance models are either lost or must be manually *migrated* to conform to the new metamodel.

The problem of evolution is not new to software engineering. In particular, databases have been dealing with schema evolution for several years. While there have been attempts to extend these techniques to model-based software [4], two characteristics of DSMLs suggest that a dedicated solution is more appropriate. First, metamodels tend to evolve in small, incremental steps, implying that a model evolution tool should focus on making these simple changes easy to specify. This also means that a large portion of the language is unaffected between versions: an ideal solution should leverage this knowledge and require only a specification for the portion that changes and automatically handle the remaining elements. On the other hand, complex changes do sometimes occur, so a mechanism for these migrations must also be available. The second point in favor of a dedicated model migration tool is that domain designers and modelers are often not software experts, which means a solution should use abstractions that are familiar to these users and avoid low level issues, such as persistence formats. Ideally, the modeler should be able to use the same abstractions to build models, metamodels and evolution specifications.

Our previous work with sequenced graph rewriting [1] provided some insight into the balance between expressiveness and ease of use. We have found that a general purpose transformation language tends to be cumbersome for the mostly minor changes present during metamodel evolution. Thus, we have designed a dedicated language called the Model Change Language (MCL) used to specify metamodel evolution in DSLs and migrate domain models. The rest of this paper describes MCL and is structured as follows. Section 2 presents further motivation and background terminology. Section 3 describes the overall design of MCL, while the implementation is presented in Section 4. Related work is found in Section 5, and we conclude in Section 6.

2. MOTIVATION AND BACKGROUND

Our primary motivation was drawn from experience with several medium and large DSMLs that continually evolved. The large number of existing instance models made manual migration impractical. For very simple language changes, such as element renamings, we found that XSLT was an acceptable solution. [11] describes a language capable of generating XSL transforms that are applied sequentially, which increases the expressiveness of the evolution, but requires the user to define the control structure and order of evaluation explicitly. Additionally, we occasionally faced more complex changes, for which XSLT was not sufficient. For these changes, our graph transformation language, GReAT [1], provided a powerful alternative, but its model migration specifications were too verbose for two primary reasons. First, when a metamodel element changes, the migration rule should be applied to all instance model elements of that type, regardless of where they are located in the model hierarchy. Second, metamodels tend to evolve in small, incremental steps, in which the majority of the elements stay the same. Together, these two points imply that a model migration tool should:

1. Contain a default traversal algorithm.

2. Automatically handle non-changed elements.

We incorporated both of these ideas into our design. Our essential hypothesis is that evolutionary changes on the modeling language will be reflected as changes on the metamodel. When the modeling language is evolved, the language designer has to modify the metamodel that will now define the new version of the language. The key observation here is that metamodel changes are explicit, and these changes are used to automatically derive the algorithm to migrate the models in the old modeling paradigm to the models compatible with the new version of the paradigm. We make an essential assumption: changes performed on the metamodel are known and well-defined, and all these changes are expressed in an appropriate language. We designed such a language, which we call the Model Change Language (MCL). We now briefly describe relevant background terminology.

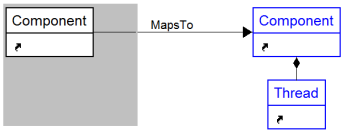
A metamodel $\mathbf{M}_L$ defines a modeling language $\boldsymbol{L}$ by defining its abstract syntax, concrete syntax, well-formedness rules, and dynamic semantics [1]. Here, we are focusing on the abstract syntax of the modeling paradigm. There are various techniques for specifying the abstract syntax for modeling languages, and the most widely used is the Meta-object Facility (MOF) [10], but for clarity here we will use UML class diagrams. The examples in this paper use UML class diagrams with stereotypes indicative of the role of the element, such as *Model* (a container), *Atom* (an atomic model element) or *Connection* (an association class) - but they may be understood as simple UML classes. Note that the actual models can be viewed as object diagrams that are compliant with the UML class diagram of the metamodel.

3. THE MODEL CHANGE LANGUAGE

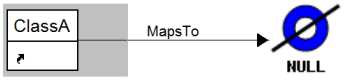
The Model Change Language (MCL) defines a set of idioms and a composition approach for the specification of the migration rules. The MCL also includes the UML class diagrams describing both the versions of the metamodel being evolved, and the migration rules may directly include classes and relations in these metamodels. MCL was defined using a MOF-compliant metamodel. For space reasons we cannot show the entire metamodel, rather we introduce the language through examples. Note that MCL uses the metamodel of the base metamodeling language, and MCL diagrams model relationships between metamodel elements. For a more in-depth look at MCL, please see [2].

The basic pattern that describes a metamodel change, and the required model migration, consists of an LHS element from the old metamodel, an RHS element from the new metamodel, and a *MapsTo* relation between them (stating that the LHS type has "evolved" into the RHS type). The pattern may be extended by including other node types and edges into the migration rule. The node at the left of the *MapsTo* forms the context, which is fixed by a depth first traversal explained in Section 4. The rest of the pattern is matched based on this context. The *WasMappedTo* link in the pattern is used to match a node that was previously migrated by an earlier migration rule. 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.

The MCL rules can be used to specify most of the common



(a) Adding a new element



(b) Deleting an element

Figure 1: MCL rules for adding and deleting elements

metamodel evolution cases, and automate the migration of instance models necessitated by the evolution of the metamodel. The core syntax and semantics is rather simple, but for pragmatic purposes higher-level constructs were needed to describe the migration. We have identified a number of categories for metamodel changes based on how metamodels are likely to evolve and created a set of MCL *idioms* to address these cases. These idioms may also be composed together to address more complex migration cases. We will describe a number of these idioms next. We first introduce the representative patterns.

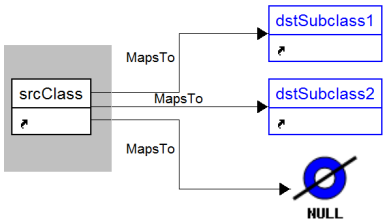
3.1 Adding Elements

A metamodel 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 are not affected by the new addition, and will continue to be conformal to the new language, except in certain cases. If the newly added element holds some model information within a different element in the old version of the metamodel, the information must be appropriately preserved in the migrated models. In fact, this falls under the category of "modification" of representation, and is described further below.

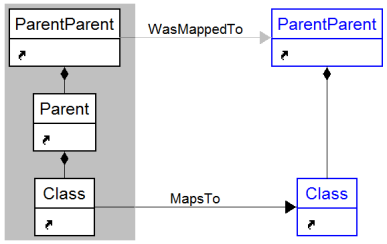
If the newly added element plays a role in the well-formedness requirements, then the old models will no longer be well formed. The migration language must allow the migration of such models to make them well formed in the new metamodel. For instance, suppose that the domain designer adds a new model 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 Fig. 1(a). The LHS or 'old' portion of the MCL rule is shown in a greyed rectangle for clarity in this and all subsequent figures.

3.2 Deleting Elements

Another typical metamodel change is 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" primitive in MCL, as shown in Fig. 1(b). This rule states that all instances of *ClassA* in the model are to be



(a) MCL rule for subclasses



(b) Changing containment hierarchy

Figure 2: MCL Rules for Subclasses and Hierarchy

removed. Removal of an object may also result in the loss of some other associations or contained objects.

3.3 Modifying Elements

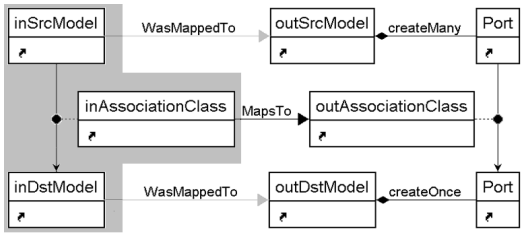
The most common change to a metamodel is the modification of certain entities, such as the names of classes or their attributes. The basic MapsTo relation suffices to specify this change. The mapping of related objects is not affected by this rule. If other related items have also changed in the metamodel,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

Another type of modification in the metamodel is adding new sub-types to a class. In this case, we may want to migrate the class's instances to an instance of one of its sub-types. Fig. 2(a) 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 Fig. 2(a) 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.

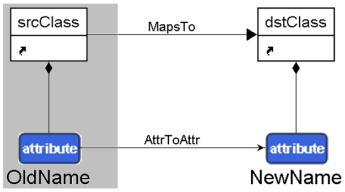
3.4 Local Structural Modifications

Some more complex evolution cases occur when changes in the metamodel require a change in the structure of the old models to make them conformant to the new metamodel. Consider a metamodel with a three level containment hierarchy, with a type *Class* contained in *Parent*, and *Parent* contained in *ParentParent*. Suppose that this metamodel 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 Fig. 2(b) (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.



(a) Rerouting associations



(b) Migrating attributes

Figure 3: MCL Rules for Associations and Attributes

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. Fig. 3(a) shows a more complex 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 perform type conversions or other operations to obtain the new value of the attribute in the migrated model. Fig. 3(b) shows an MCL rule for migrating attributes.

In addition to the idioms listed so far, the tool suite for model migration supports additional idioms to handle other common migration cases. Fig. 4 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. The idiom for deleting an attribute is similar to the case of deleting classes and is not shown due to space constraints. Fig. 5(a) 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.

Fig. 5(b) shows an idiom for changing a containment relationship in the metamodel. This is a variation of the idiom shown earlier in Fig. 2(b), for a more generic case. This idiom also introduces a generic primitive called "Navigate". It can be used to locate objects in the instance model by following a navigation condition, which is an iterator over the graph. Starting from the object on the left end of the Navigate link, this object is used to determine the new parent in the migrated model. Fig. 6(a) 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 (shown separated by a dashed line). The migration rule can encode a command that will set the value of the attribute based on its original type. Fig. 6(b) 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 destination side.

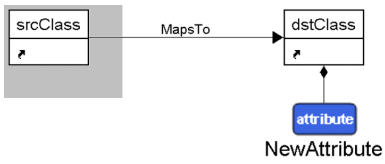
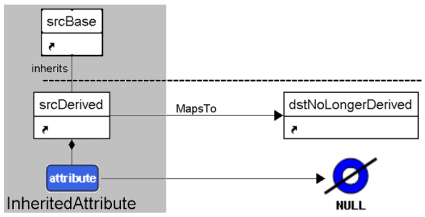
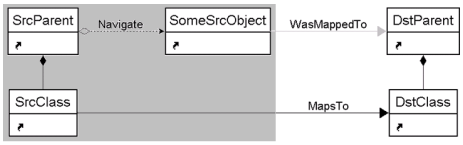


Figure 4: MCL Rule for Attribute Addition



(a) Delete inheritance relationship



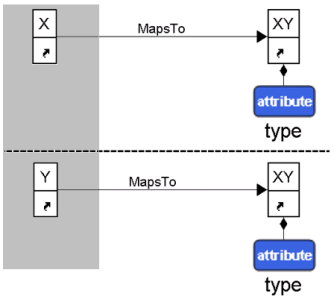
(b) Change containment relationship

Figure 5: MCL Rules for Inheritance and Containment

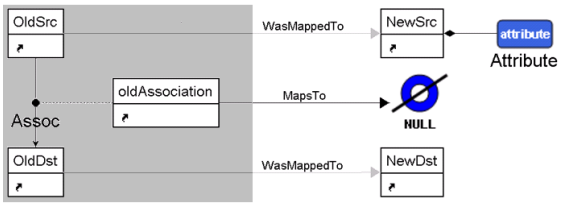
These idioms may also be composed together to accomplish more complex evolutions. The following section presents the details of the MCL implementation.

4. IMPLEMENTATION OF MCL

Our model migration approach consists of three aspects. The first is a complete tree rewrite based on the depth-first traversal of the input model. The second aspect consists of a



(a) Merge classes



(b) Replace association with attribute

Figure 6: MCL Rules for Merging Classes and Replacing Associations

```
Depth-first traversal algorithm
ModelMigrate( oldModel )
  call TraverseTree( oldModel.RootFolder )
  if delayQueue.length != 0 then call ProcessQueue

TraverseTree( node )
  call MigrateNode( node )
  foreach childnode in node.children do
    call TraverseTree( childnode )
```

set of migration rules that specify the rewriting of the model elements (nodes) whose type has changed in the metamodel. The third aspect is a delayed rewrite approach that uses lazy evaluation for the rewriting of nodes that cannot be immediately processed. These are explained in detail below. The migration algorithm maintains a map of the node instances migrated so far (mapping a node in the old version model to its corresponding node in the migrated model), which we call the *ImageTable*, allowing the use of previously mapped nodes in other migration rules. We found that this approach best suited our needs for model evolution, as it simplified the specification and execution of the migration rules. The pattern matching effort required by these rules is limited, while allowing the co-existence of different versions of the model.

Depth-first traversal and rewrite. The tree rewrite starts at the root node (*RootFolder*) of the input model, creating a corresponding root node for the migrated (output) model. It follows a depth-first traversal of the input model based on its containment relationships, while creating the output model in the same order. Each node is migrated either by (1) a default migration which creates a 'copy' in the output model, or by (2) executing the migration rule specified for its type. Some migration rules may not be executed immediately and are queued and handled later.

Migration Rules. Typically, when a metamodel evolves, only a small number of the types and relations defined in the metamodel are changed. For the unchanged types, the

default 'copy' operation suffices in the tree rewrite described above. For the cases where the type has changed, a migration rule is used to specify the actions necessary to migrate an instance of that type into the output model.

Migration rules are specified using MCL as described above in section 3. An MCL rule is specified for a particular (node) type in the metamodel, and consists of a pattern which may involve other node types, a *MapsTo* relation that specifies how the node type is migrated, optional *WasMappedTo* relation(s), and additional imperative *commands* and *conditions* to control node creation. The *commands* are imperative actions executed during node creation, and *conditions* are Boolean expressions that control whether the migration is allowed to happen. The *WasMappedTo* relation specifies a node instance in the output model that was previously migrated corresponding to a certain node instance in the input model (maintained in the *ImageTable*). The migration of a node begins by finding a migration rule for that node type. With the node instance as context, the rest of the rule elements are matched by matching the appropriate nodes in the input model. If the match is not successful because the *WasMappedTo* relation is not satisfied (yet), the node is added to a queue to be processed later. Otherwise, the specified node is created in the migrated (output) model, and the depth-first traversal continues.

Migration algorithm for a Node

```
MigrateNode( node )
  let rule = FindMigrationRule( node.type )
  if rule == null then call DefaultMigrate( node )
  else call ExecuteRule( rule, node )

FindMigrationRule( nodetype )
  find rule in ruleSet where mapsTo.LHS.type = nodetype
  if found return rule else return null

DefaultMigrate( node )
  let newtype = node.type
  if newtype not in newMeta.types
    then throw TypeNotFoundError
  let oldParent = node.parent
  let newParent = ImageTable.findNewNode( oldParent )
  if newParent == null then
    call delayQueue.addNode( node )
    return
  let newnode = CreateNode( newtype, newParent )
  call CopyAttributes( node, newnode )
  call ImageTable.addImage( node, newnode )

ExecuteRule( rule, node )
  let matchResult = MatchRulePattern( rule, node )
  if matchResult == true then //Match succeeded
    if Eval( rule.condition ) == false then return //Can't apply
    let newtype = rule.mapsTo.RHS.type
    if rule.newParent == null then
      let oldParent = node.parent
      let newParent = ImageTable.findNewNode( oldParent )
    let newnode = CreateNode( newtype, newParent )
    call CopyAttributes( node, newnode )
    call Eval( rule.command )
    call ImageTable.addImage( node, newnode )
  else // Match failed, queue node
    call delayQueue.addNode( node )
```

Queuing and Delayed Rewrite. In certain cases, such as a migration rule that depends on a mapping for another node which has not yet been migrated, the migration for that node cannot be executed. But it may be possible to execute the migration after some other migration rules have been executed. We use a delayed rewrite approach to handle these cases, by queuing the nodes for which the migration cannot be immediately effected. The listing below describes this algorithm. After completing the first pass of the depth-first traversal, the queued nodes are processed by calling *ProcessQueue*. Nodes are removed from the queue (in FIFO order), and migration is attempted again. If *MigrateNode* fails, the node is added back at the end of the queue. If the length of the queue has changed after one pass, *ProcessQueue* is called again. The algorithm terminates when the queue is empty, or when a complete pass of the queue has not changed the queue.

Delayed rewrite algorithm

```
ProcessQueue( )
  let qLength = delayQueue.length
  if qLength == 0 return
  for index = 1 to qLength
    let node = delayQueue.removeTopNode
    call MigrateNode( node )
  if delayQueue.length < qLength then call ProcessQueue
```

5. RELATED WORK

Our work on model-migration has its origins in techniques for database schema evolution. More recently, though, even traditional programming language evolution has been shown to share many features with model migration. Drawing from experience in very large scale software evolution, [6] uses several examples to draw analogies between tradition programming language evolution and meta-model and model co-evolution. [3] also outlines parallels between meta-model and model co-evolution with several other research areas, including API versioning.

Using two industrial meta-models to analyze the types of common changes that occur during meta-model evolution, [9] gives a list of four major requirements that a model migration tool must fulfill in order to be considered effective: (1) Reuse of migration knowledge, (2) Expressive, custom migrations, (3) Modularity, and (4) Maintaining migration history. The first, reusing migration knowledge, is accomplished by the main MCL algorithm: meta-model independent changes are automatically deduced and migration code is automatically generated. Expressive, custom migrations are accomplished in MCL by (1) using the meta-models directly to describe the changes, and (2) allowing the user to write domain-specific code with a well-defined API. Our MCL tool also meets the last two requirements of [9]: MCL is modular in the sense that the specification of one migration rule does not affect other migration rules, and the history of the meta-model changes in persistent and available to migrate models at any point in time.

[5] performs model migration by first examining a difference model that records the evolution of the meta-model, and then producing ATL code that performs the model migration. Their tool uses the difference model to derive two model transformations in ATL: one for automatically resolvable changes, and one for unresolvable changes. MCL uses a difference model explicitly defined by the user, and uses its core algorithm to automatically deduce and resolve the breaking resolvable changes. Changes classified as breaking and unresolvable are also specified directly in the difference model, which makes dealing with unresolvable changes straightforward: the user defines a migration rule using a graphical notation that incorporates the two versions of the meta-model and uses a domain-specific C++ API for tasks such as querying and setting attribute values. In [5], the user

has to refine ATL transformation rules directly in order to deal with unresolvable changes.

[7] describes the benefits of using a comparison algorithm for automatically detecting the changes between two versions of a meta-model, but says they cannot use this approach because they use Ecore-based meta-models, which do not support unique identifiers, a feature needed by their approach. Rather than have the changes between meta-model versions defined explicitly by the user, they slightly modify the ChangeRecorder facility in the EMF tool set and use this to capture the changes as the user edits the meta-model. Their migration tool then generates a model migration in the Epsilon Transformation Language (ETL). In the case that there are meta-model changes other than renamings, user written code in ETL to facilitate these changes cannot currently be linked with the ETL code generated by their migration tool. In contrast to this, MCL allows the user to define complex migration rules with a straightforward graphical syntax, and then generates migration code to handle these rules and links it with the code produced by the main MCL algorithm.

[8] presents a language called COPE that allows a model migration to be decomposed into modular pieces. They note that because meta-model changes are often small, using endogenous model transformation techniques (i.e., the meta-models of the input and output models of the transformation are exactly the same) can be beneficial, even though the two meta-models are not identical in the general model migration problem. This use of endogenous techniques to provide a default migration rule for elements that do not change between meta-model versions is exactly what is done in the core MCL algorithm. However, in [8], the meta-model changes must be specified programmatically, as opposed to MCL, in which the meta-model changes are defined using a straightforward graphical syntax.

Rather than manually changing meta-models, the work in [13] proposes the use of QVT relations for evolving meta-models and raises the issue of combining this with a method for co-adapting models. While this is an interesting idea, our MCL language uses an explicit change language to describe meta-model changes rather than model transformations.

Although not focused on meta-model or model evolution, the work in [12] is similar to our approach. The authors perform the automatic generation of a semantic analysis model from a domain-specific visual language using a special "correspondence" model called a *meta-model triple*. The connections provided by the meta-model triple perform a similar role as the *MapsTo* and *WasMappedTo* links in MCL.

6. CONCLUSIONS

We have presented the Model Change Language (MCL), our language for specifying metamodel evolution and automatically generating the corresponding model migration. MCL requires the specification of only the evolved parts of a meta-model and automatically handles the persistent parts. The specification is done using the metamodels of the original and evolved language, which allows domain experts to use the same abstractions for specifying both metamodels and their evolution. Our implementation produces executable code to perform model migration from the evolution specification and has been integrated with our Model-Integrated Computing (MIC) metaprogrammable toolsuite and tested on a number DSML evolution examples of medium complexity. These test metamodels typically consisted of 50-100 elements, and the number of migration rules were on the order of 5-10. The examples were used in proof-of-concept demonstrations where savings in development effort were measured with promising results.

The model migration problem is an essential one for model-driven development and tooling, and there are several challenging problems remaining in this area. Efficiency of the migration code is of paramount importance, especially on large-scale models. The migration idioms that we have targeted were based on our past experience, but it appears 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.

Acknowledgment. This work was sponsored by DARPA, under its Software Producibility Program. The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA or the US government.

7. REFERENCES

[1] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, and G. 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*, 2002.

[2] D. Balasubramanian, C. vanBuskirk, G. Karsai, A. Narayanan, S. Neema, B. Ness, and F. Shi. Evolving paradigms and models in multi-paradigm modeling. Technical report, Institute for Software Integrated Systems, 2008.

[3] P. Bell. Automated transformation of statements within evolving domain specific languages. In *7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[4] P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD 07*, 2007.

[5] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC*, pages 222–231, 2008.

[6] J.-M. Favre. Meta-models and Models Co-Evolution in the 3D Software Space. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM*, 2003.

[7] B. Gruschko, D. S. Kolovos, and R. F. Paige. Towards Synchronizing Models with Evolving Metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution (MODSE)*, 2007.

[8] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE: A Language for the Coupled Evolution of Metamodels and Models. In *MCCM Workshop at MoDELS*, 2009.

[9] M. Herrmannsdoerfer, S. Benz, and E. Jürgens.

Automatability of Coupled Evolution of Metamodels and Models in Practice. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS*, pages 645–659, 2008.

[10] MOF. Meta-Object Facility: Standards available from Object Management Group.

[11] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291 – 307, 2004. Domain-Specific Modeling with Visual Languages.

[12] H. Vangheluwe and J. de Lara. Automatic generation of model-to-model transformations from rule-based specifications of operational semantics. In *7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[13] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference*, pages 600–624, 2007.