Model Migration through Visual Modeling

Jonathan Sprinkle
Vanderbilt University
jonathan.sprinkle@alumni.vanderbilt.edu

Gabor Karsai
Vanderbilt University
gabor@vuse.vanderbilt.edu

Abstract: The true value of domain-specific modeling is found not in a domain-specific modeling language (DSML) but rather in the models that are created using that DSML. Changes to a physical system can be modeled, and the resulting executable model then is a working version of the physical system. Unfortunately, if the model of the domain—or metamodel—is changed, all models that were defined using that metamodel may require maintenance to have semantics that correctly represent the existing system. This paper introduces a visual modeling environment that allows for the evolution of modeling environments using XSL stylesheets as the execution language.

1. Introduction

Domain-specific modeling languages (DSMLs) have a unique advantage over general-purpose programming languages: the ability to encode domain semantics into the language [9]. Thus, the load on the programmer (i.e., the domain-specific modeler) is lessened because program information unrelated to the specification of the system can be generated. One distinct disadvantage of this approach, however, is that modifications to the DSML modify the language in which the domain models (that is, models created in the DSML) were originally intended to exist—meaning that the domain models may not be valid "programs" in the new DSML. Another possibility is that the domain models *are* valid in the new DSML, but that the domain semantics encoded in the DSML have changed, thus requiring that the domain models be modified to work "correctly" in the new DSML.

These issues are encountered whenever a DSML is modified and there exists a collection of domain models defined for that DSML. The evolution of the domain models to the new DSML (referred to in this paper as *model migration*) is highly dependent on the particular changes made to the DSML, and as such requires not a single solution for all model migration problems, but rather a language designed for the *domain* of all model migration problems.

This paper presents one such language, which is designed to migrate models created using the MetaGME metamodeling language [1]. The actual migration is performed by XSL stylesheets that are generated by the presented visual language, and may be executed by any XSL engine.

2. Background

2.1. Domain-Specific Modeling

A domain-specific language is used by a domain expert instead of low-level code to create systems in the domain. The domain-specific language decreases construction time and provides a specialized interface for managing the domain concepts. This approach can be used to apply a sort of "golden rule" to drive the development of a domain-specific language: *the size of the change in requirements should be proportional to the size of the change in the implementation.*

A technique called *metamodeling* can be used to easily—yet precisely—describe the syntax and static semantics (the well-formedness rules) of a modeling language. The artifact of the metamodeling process—called the metamodel—is generally retained in an object database

for later manipulation, and can be evolved to create a new version of the DSML. *Domain models* are instances of system types, and are used to represent the structural or behavioral aspects of an existing system.

Before metamodeling became a widely used method for creating DSMLs [2][3][4][5][7][8][10], most such languages were not apt to evolve quickly. Early DSMLs were usually not fully implemented and deployed until vetted by domain experts, because of the complexity associated with updating the DSML. Metamodeling provided the ability to rapidly develop *and change* the domain-specific language. However, this ability to rapidly create and deploy modeling languages came at a cost: changes to the language also made obsolete any *domain models* (i.e., models created using the DSML) created before its update.

2.2. Domain Evolution

Despite best practices in design, accepted standards and interfaces, domains can—and do—change in their essential definition over the passage of time. That is, eventually the entire family of systems changes in some way, and the entities and principles of the domain that were present in the original design may change or may be removed for reasons outside the control of the language. The evolution of the family of systems is called *domain evolution*. In this case, the DSML—as it is a model of the domain—must be updated to reflect these changes. The system requires maintenance on the metamodel level, resulting in an evolution of the domain-specific modeling language [6].

The valuable portion of a DSML is not the modeling language itself, which can be generated, but the body of models that is built up by the modeler [13]. When domain evolution is required the model databases created with the old DSML may no longer be an accurate description of the system they once helped implement, due to the changes required in the semantic translator and/or semantic domain.

2.3. Model Migration

The evolution of models created using formal methods is simpler than the evolution of low-level software due to the restricted nature of most formal methods used to create languages. DSMLs are defined using metamodels; in essence, a metamodel acts as a schema for a domain-specific modeling language. The evolution of a DSML can be performed by creating an algorithm that is composed of elements from the language metamodel. The execution of this algorithm on existing domain models to transform them into domain models that are correct in the evolved domain is called *model migration*. In short, model migration is a way to carry out domain evolution for a DSML.

Domain evolution, as its name suggests, is typically evolutionary rather than revolutionary in its modifications. Model migration should reflect this by focusing on the differences between the two DSMLs rather than their similarities. In other words, the model migration domain should follow the "golden rule" in the following way: that small changes made to a metamodel should require only a small specification to migrate any domain models. That is, the size of the model migration specification should be related to the size of the change to the metamodel, and not the size of the metamodels themselves.

3. A Domain Evolution Framework (DEF)

A meta-metamodel can be used to create a metamodel that describes any modeling language. That being said, not all meta-metamodels provide a convenient representation of the same abstractions, and some meta-metamodels—while very convenient for describing some abstractions—are tedious or counter-intuitive for others. This requires the concession (or, per-

haps conjecture) that there is not now, nor will there ever be, one ubiquitous meta-metamodel. The consequence of this conjecture is that a *domain evolution tool* is required for each meta-metamodel. Although the complete set of domain evolution tools (one for each meta-metamodel) does not contain any identical tools, many of the tools will contain similar properties.

A *domain evolution framework* (DEF) is a generic representation of the fundamental properties and algorithms that are contained in each and every domain evolution tool. As a generic representation it provides some parameters for instantiation, and those parameters are the meta-metamodel and transformation engine used to instantiate a particular domain evolution tool. When that domain evolution tool is used to evolve domain models from one metamodel to another, then a *domain evolution specification* has been created as an artifact. The domain evolution specification must be specified in at least two dimensions: namely, the patterns in domain model *D* that are to be recognized and how those patterns should appear in the transformed model, *D'*.

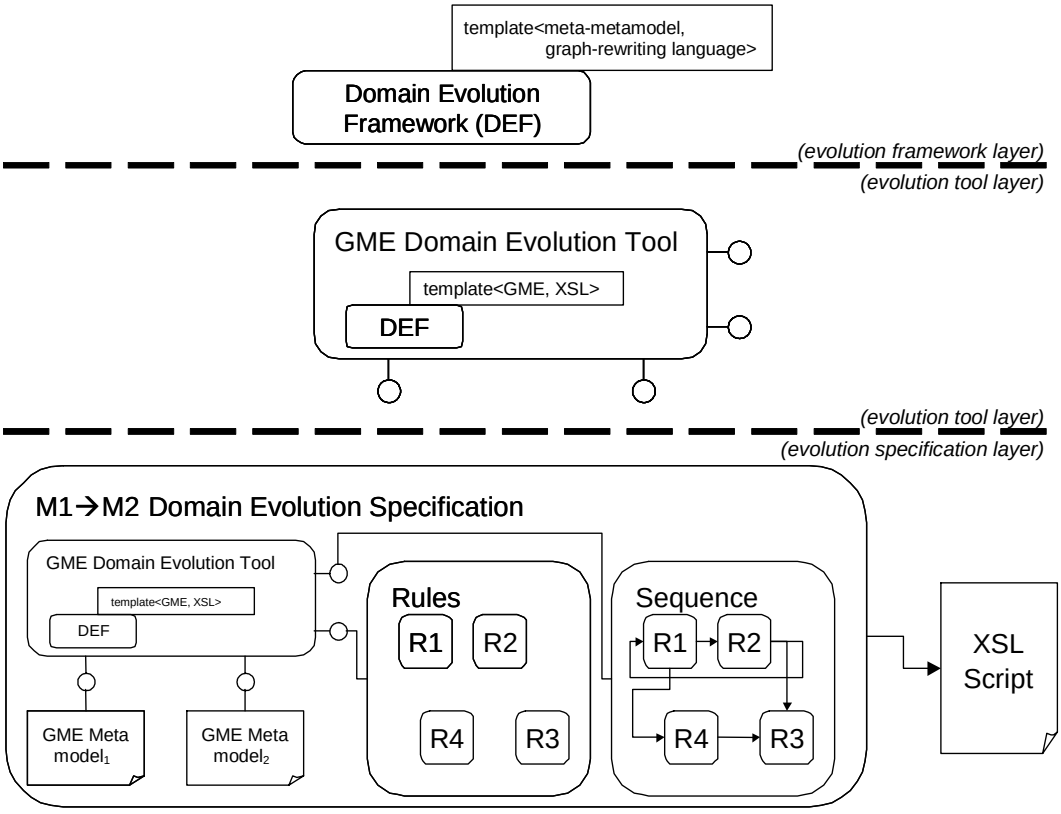


Figure 1. Layers of the domain evolution tool for MetaGME and XSL. Note that the final domain evolution specification is one particular evolution of domain models from GME Metamodel 1 (M1) to GME Metamodel 2 (M2)

Figure 1 shows the three layers for a particular instantiation of a domain evolution tool for the GME meta-metamodel and XSL graph transformations. The layers range from most generic at the top (the framework) to least generic (the specification that generates an XSL script). The *evolution framework layer*, at the top, contains the common information found in all domain evolution tools, and provides an interface to be customized by meta-metamodel and graph-rewriting language. In this sense, the evolution framework layer is quite similar to a template class definition in C++. The *evolution tool layer*, in the center, has tools that are specific instances of the DEF. Shown in the figure is an example tool that parameterizes the

DEF using the GME meta-metamodel and the XSL graph-rewriting language. The domain evolution tool is an environment usable by a modeler who is performing model migration.

4. The MetaGME/XSL domain evolution tool

The DEF is a collection of the common architecture components, user interfaces, programming interfaces of any tool that could be used to perform domain evolution. Any specific instance of the framework described has four distinct parts: (1)(2) the two DSML definitions (as defined by a common meta-metamodel), (3) transform rules and their sequencing, and (4) the item types that can be matched along with the types of mappings that can be performed between matched items. Figure 2 shows an overview of this framework, with the four components identified.

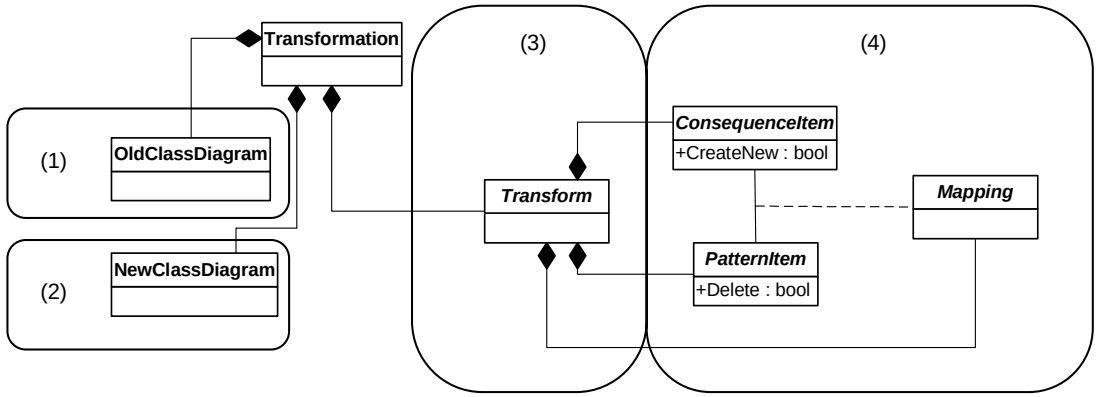


Figure 2. Simplified overview of the DEF. Comparison of this figure with Figure 1 shows the interfaces required for a domain evolution specification to be created with the framework

4.1. Transform types

In general a model migration algorithm is made up of sequenced `Transforms`. `Transforms` are used to describe the specific differences between metamodels, but not all `Transforms` carry the same semantics. In addition to their ability to describe the differences between metamodels, `Transforms` should also be capable of specifying where they should execute in control flow – their sequence. `Transforms` are not necessarily required to be sequenced.

There are three possible types of `Transforms`, explained below.

- `Rule` – has side effects, and is sequenced. It may also contain a sequenced object (either another `Rule` or a `Test`) in order to allow hierarchically structured `Transforms`.
- `Test` – is sequenced, but does not have side effects. It is used to make decisions in the sequence of `Transforms` by executing the `Cases` and other `Tests` it contains. Although it does not have side effects, the `Cases` it contains may.
- `Case` – is not sequenced, but has side effects. It is a special kind of `Rule` that behaves the same, but without the ability to be sequenced or contain any `Transforms`. Its purpose is to perform its mapping (if any) and return a boolean result if its pattern is matched. The value of the result is used for decision making in the execution of the model migration algorithm.

4.2. Legal Items

The replacement of syntax patterns from the old domain with patterns in the new domain is the basis of domain evolution. The types of objects that can be used to create these patterns are termed *legal items*.

Legal items are identified as patterns, matched, and upon successful match transformed through *mappings* into objects in the new DSML (consequences, as opposed to patterns). The types for mapping associations form a fundamental set of operations, in similar fashion to the fundamental set of string replacement operations (i.e., insert, concatenate, delete, replace). The determination of the fundamental set of replacement operators for models is a significant problem in its own right. Textual replacement has only one "association" – position – while object replacement in a graph can have as many associations as are permitted for that type of object by the metamodel. This paper limits that set of operators to `CreateNew`, `CreateWithin`, `Becomes`, and `Delete` (analogous to insert, concatenate, replace and delete for textual manipulation).

- `CreateNew` – an attribute of a `ConsequenceItem`. If the LHS is matched then any object with this attribute set to true will be created. There must be a `CreateWithin` association in which this `ConsequenceItem` is either the source or destination.
- `CreateWithin` – a binary association between a parent `LegalItem` (destination) and child `LegalItem` (source). The parent and child may be either a pattern or consequence, but at least one of them must be a consequence object. An object taking part in this association may not take part in a `Becomes` association, and may not have its `Delete` attribute set to true if it is a `PatternItem`.
- `Becomes` – a binary association between a source `PatternItem and a destination ConsequenceItem`. The source may not have its `Delete` attribute set to true, and the destination object may not have its `CreateNew` attribute set to true.
- `Delete` – an enumerated attribute of a `PatternItem`. If this attribute is set to `ObjectOnly`, and the `PatternItem` is part of a match then that item will be deleted (i.e., will not be copied directly into the output model) and any children of the object will be preserved by default and placed in the parent of the deleted object. If this attribute is set to `ObjectAndChildren` then all children will be recursively deleted.

The assumption that allows for this limited set of mappings relies on the definition of additional associations in a metamodel, and is outside the scope of this paper, but explained further in [11].

4.3. Model of Computation

Because the DEF must be "translatable" into any full-featured graph-rewriting language it is necessary for the framework to have its own defined model of computation (MoC). It is beyond the scope of this paper to fully define the MoC of the framework, so these short descriptors are given.

- To specify the execution dependency of rules, they may be sequenced
- Rules that are not sequenced are not guaranteed to be executed in a particular order
- One initial sequence may be specified, to guarantee that this sequence happens before any other rules
- Control flow can be specified using the `Test` objects

4.4. Visualization of the Language

The rules that make up the overall model migration algorithm are containers for patterns that describe the replacement/creation/deletion of objects such that the models can be migrated. An example is shown in Figure 3. The ProcessSignal, Port, and Connection objects are used to establish a context for transformation. Given that the pattern is found in a source model, the appropriate transformation (in this case, changing the type of the Port objects to Input) is carried out. The next section explains some of the details of this transformation.

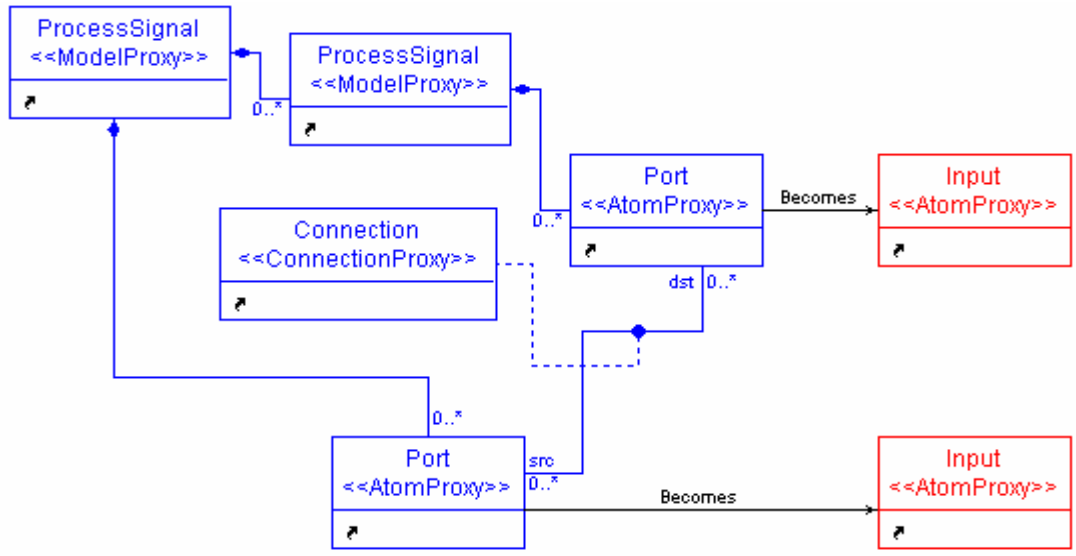


Figure 3. An example visual pattern that describes the transformation of Port objects to objects of type Input.

5. Mapping the visual language into XSL stylesheets

A basic study of the XML persistence format to store GME domain models allows for the determination of the format for generated XSL stylesheets. It is possible to set forth a mapping from the patterns described using the model migration interface to patterns in XSL—or more accurately, in XPath—that will uniquely identify the domain models to be migrated, and map them into appropriate new concepts in the new DSML.

Objects not specified to be mapped are copied isomorphically into the output document using an isomorphic transform (omitted in this document, for brevity). For an in-depth discussion of mapping translations, this transform, and the XSL model of computation, refer to [11][12].

5.1. Pattern matching and mappings

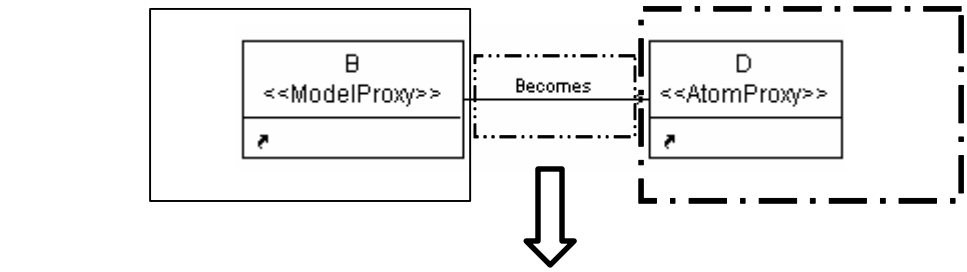
For each mapping in the model migration specification a named template is generated in an XSL stylesheet. An example named template (and its corresponding diagram) of the `Becomes` mapping is given in Figure 4. Note that each named template is encoded with the change that it performs (in this case `B` becomes a `D`) and it is uniquely named to avoid conflicts with multiple changes of the same type. If containment or association context is required for a particular match, the additional criteria is inserted into the "test" portion of the XSL template. Containers and dotted lines are added to the graphical specification and its generated XSL template to call attention to which portions of the XSL are generated from which areas of the graphical specification.

An analogous method is used when performing `CreateWithin` (and consequently `CreateNew`) except that the template would be called from within the template body of the object in which the consequence was to be created. When using the `Delete` mapping the template body is empty.

5.2. Execution model and sequencing

The DEF MoC is synthesized through the creation of a different stylesheet for each `Transform` in the transformation, and passing an XML artifact to each stylesheet, keeping only the final XML artifact. This solution has guaranteed predictability of execution, and an

easily composable nature. Figure 5 shows the sequencing execution model. Tests and Cases are implemented in an analogous manner, see [11].




```
<xsl:template match="model[@kind='B']">
 <xsl:variable name="Becomes_focus_B" select="current()"/>
 <xsl:choose>
  <xsl:when test="current()[$Becomes_focus_B]">
   <xsl:call-template name="BBecomesD_2"/>
  </xsl:when>
  <xsl:otherwise>
   <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
   </xsl:copy>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>
```

```
<xsl:template name="BBecomesD_2">
 <xsl:element name="atom">
  <xsl:apply-templates select="@*"/>
  <xsl:attribute name="kind">D</xsl:attribute>
  <xsl:comment>transformed using template 'BBecomesD_2'</xsl:comment>
  <xsl:apply-templates select="node()"/>
 </xsl:element>
</xsl:template>
```

Figure 4. Mapping transformation specified as a named template

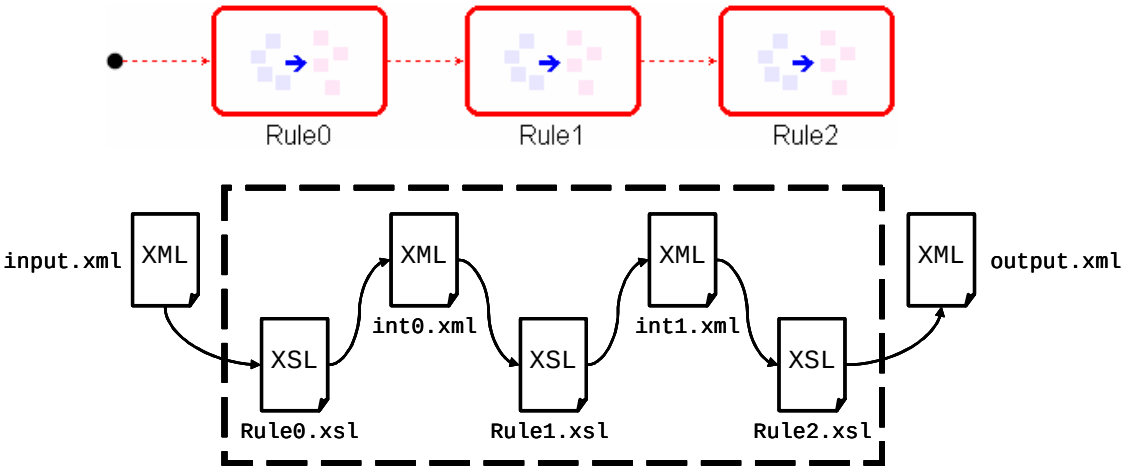


Figure 5. Example transformation exemplifying the translation into a sequenced execution

6. Conclusions

While there are other visual programming languages available that can be used to achieve the same ends that the presented language does (see [11]), none are specific to the problem of model migration, and none use elements from the formal metamodels of the domains in order to specify how the models are to be migrated.

This research has been a benefit to the process of the evolution of domain specific modeling languages by providing a language developed expressly for the purpose of migrating domain models. It was developed based on a domain evolution framework that is independent of metamodeling language and graph transformation implementation, so it has an interface that is designed for model migration rather than XSL stylesheets. The language allows the user to visually program the evolution algorithm, and utilizes the types found in the metamodels of the two domains as language primitives.

Most importantly the language satisfies the "golden rule" by providing a way to evolve domain-specific modeling languages by specifying a model migration algorithm that is proportional to the size of the evolution rather than the size of the metamodel.

This research was performed under the sponsorship of the Defense Advanced Research Projects Agency, Information Exploitation Office, Model-Based Integration of Embedded Systems project, under contract number #F30602-00-1-0580, and also the NSF ITR on "Foundations of Hybrid and Embedded Software Systems".

7. References

[1] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, "Composing Domain-Specific Design Environments", IEEE Computer, pp. 44-51, November, 2001.

[2] J. Sztipanovits, et al., "MULTIGRAPH: An Architecture for Model-Integrated Computing", *Proc. IEEE International Conference on Engineering of Complex Computer Systems*, pp. 361-368, 1995.

[3] OMG Unified Modeling Language Specification, ver. 1.4, Object Management Group, et al., September 2001.

[4] R. Lemesle, "Transformation Rules Based on Meta-modeling", Proceedings of the Second International Enterprise Distributed Object Computing Workshop, San Diego, November 1998.

[5] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", *Proceedings of the IEEE ECBS'99 Conference*, pp. 68-74, Nashville, Tennessee, April, 1999.

[6] G. Nordstrom, J. Sztipanovits, G. Karsai, "Metalevel Extension of the MultiGraph Architecture", *Proceedings of the IEEE ECBS'98 Conference*, pp. 61-68, Jerusalem, Israel, April 1998.

[7] The Domain Modeling Environment (DoME). http://www.htc.honeywell.com/dome/index.htm

[8] MetaEdit+, Domain specific modeling tool. http://www.metacase.com/

[9] D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics, and All That Stuff. Part I: The Basic Stuff", Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute Of Science, Rehovot, Israel.

[10] G. Karsai, G. Nordstrom, A. Ledeczi, J. Sztipanovits, "Specifying Graphical Modeling Systems Using Constraint-based Metamodels", *Proceeding of the IEEE Symposium on Computer Aided Control System Design*, Anchorage, AK, 2000.

[11] J. Sprinkle, "Metamodel Driven Model Migration", Ph.D. Dissertation, pp. 104-106, Vanderbilt University, August, 2003.

[12] M. Kay, *XSLT – Programmers Reference*, Wrox Press Ltd., 2nd Edition, 2002.

[13] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, "On the use of Graph Transformations in the Formal Specification of Computer-Based Systems", *IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, pp. 19-27, Huntsville, Alabama, April 9, 2003.