

A Paradigm for Teaching Modeling Environment Design*

Jonathan Sprinkle
University of California,
Berkeley
333 Cory Hall
Berkeley, CA 94720
sprinkle@eecs.berkeley.edu

James Davis
Institute for Software
Integrated Systems
Vanderbilt University
2015 Terrace Place
Nashville, TN 37203
james.davis@vanderbilt.edu

Greg Nordstrom
DataCentric Automation
7109 Baker's Bridge Road
Brentwood, TN 37027
greg.nordstrom@rfmcentral.com

ABSTRACT

Model-Integrated Computing (MIC) is a generic term for the practice of coupling models of a complex system with the execution of that system. Although MIC has been in use for years by experts who learn its techniques in an *ad hoc* manner, it is only recently that system modeling—as a science—has begun to be taught as a subject. This paper presents a paradigm in use in a course designed specifically to teach the design of domain-specific modeling environments. The work describes how this paradigm grows throughout the course to complement the expanding nomenclature, mapping technologies, visitor and object-oriented technologies, and design decisions encountered by the students.

Categories and Subject Descriptors

[Education issues]: New courses; [Languages]: Domain-specific

General Terms

model-based design, metamodeling

1. INTRODUCTION

Model-Integrated Computing (MIC) is a methodology for the emerging discipline of system-level design [7]. Rather than translate system requirements into an executable system, system-level design emphasizes the specification of system existence as a set of formal *models*. This set of formal models is then examined by an intelligent translator to produce a formal executable system (e.g., low-level code) in the application domain.

The MIC methodology is effective in reducing system development time, as well as providing documentation intrinsically in the visualization of the system models. In ad-

*This work was sponsored in part by the NSF ITR on “Foundations of Hybrid and Embedded Software Systems”

dition, applications generated from MIC solutions are easier to evolve, since the applications are regenerated from an evolved system model—reducing implementation errors that occur in low-level coding. Well-designed MIC solutions are created with possible changes to the systems in mind—meaning that they are designed to easily evolve the applications.

However, the MIC methodology is not trivial to understand, due to the abstract nature of language specification, modeling terms, and application domain artifacts; furthermore, learning not only what MIC is but *how to* design MIC solutions well has been accomplished to date in a fashion similar to the early days of software: as an art, rather than a science. The course we describe herein is colloquially called the “MIC Course”, and its main purpose is to objectify the art of creating an MIC solution.

This paper describes the methodologies and pedagogy used in the MIC course to teach the discipline of system-level design through MIC. The next section provides some brief background on the course, as well as some definitions and information on modeling concepts and modeling environments. Next, the concepts and methodologies considered key in learning to design MIC solutions are described, followed by a section giving an example domain, and its ability to touch all of these key aspects of MIC solution design. Finally, we present our conclusions, and future research.

2. BACKGROUNDS

2.1 System-level specification

System-level design, regardless of the domain, aims at the capture the behavior of the system separate from its implementation [3]. Similar to the way that separation of concerns provides orthogonal aspects through which to view software [2, 1], system-level design treats the implementation details as orthogonal specifications to the behavior of the system—so long as those details are not functional requirements themselves (e.g., CPU cache size, operating system versions, or implementation language). However, whereas aspect-oriented programming provides different cutpoints and views of the software specification, system-level specification uses information hiding and hierarchy to specify the system.

2.2 Domain-specific modeling

Domain-specific modeling [9, 10, 11] allows for system-level specification in a language customized for a particular do-

main. An integrated development environment called the domain-specific modeling environment (DSME) captures the formal model specification language, and translator, in an application available to a domain expert.

Although DSMEs are frequently visual, they should not be confused with visual programming. On the contrary, DSMEs are highly restrictive syntaxes that choose a graphical denotational semantics that reflect the common notations (which are often informal) of a domain expert. Domain-specific modeling formalizes (to a reasonable extent) notations and semantics common to all domain-experts, in order to provide system behavior from these notations, rather than use the notation as a reference when encoding the behavior in an implementation language.

Once the language of the DSME is finalized, one or more intelligent translators is created to extract behavioral specifications (e.g., functional, performance, reliability) from the formalized notation and translate these specifications into the language of the formal executable system(s) in the application domain.

2.3 Abstract modeling concepts

All modeling formalisms can be traced back to an atomic set of abstract modeling concepts. Each of these concepts provides a key feature for abstract modeling, and the atomic concepts may be combined to provide a complex behavior of modeling concepts. The following list gives the set of abstract concepts used in this paper:

- *Module interconnection*: provides rules for connecting objects together through defined interfaces; modules contain ports, ports connect to other ports
- *Aspects*: enables multiple views of a model; used to allow models to be constructed and viewed from different “viewpoints”
- *Hierarchy*: describes the allowed encapsulation and hierarchical behavior of model objects; used to represent information hiding
- *Object association*: binary and n-ary associations among modeling objects; used to constrain the types and multiplicity of connections between objects, or to specify conditional containment
- *Specialization*: describes inheritance rules; used to indicate object refinement.

While other nomenclature may be given to this set (and certainly other sets exist) the semantics of the abstract modeling concepts remains the same: that there are basic techniques used to hide information, streamline type definitions, choose information filters, establish interfaces, and create associations between objects.

2.4 The Generic Modeling Environment

The Generic Modeling Environment (GME) [4] is an implementation of a metaconfigurable modeling environment. GME provides DSME interfaces by configuring its native

Table 1: Mapping of GME concepts onto abstract modeling concepts

Abstract Concept	GME Concept
Module Interconnection	Models containing Atomic Parts, where Atomic parts play role of interconnection ports
Multi-Aspect Modeling	Aspects
Hierarchy	Model/Atomic Part Containment
Object Association	1. Conditionalization 2. Binary Connections 3. Atomic Part- and/or Model References; References to References
Specialization (a.k.a. Inheritance)	Done at language design time

user-interface, constraint manager, and backend-interface through a *paradigm*. A paradigm in the GME parlance is a specialization of the basic modeling concepts of GME into user-defined types. A paradigm can be specified by the MetaGME [5, 8] DSME, provided with GME. MetaGME allows a modeler to specify the abstract syntax of a DSME using a stereotyped class diagram with UML-like syntax and semantics.

GME provides concrete object types that implement the behavior of abstract modeling concepts. These concrete object types are,

- *Model*: can participate in associations, and contain other Models, Atoms, References, Sets, or Connections (MARSCs) through hierarchical association
- *Atom*: can participate in associations (i.e., a model without hierarchy)
- *Reference*: a unary “pointer” to a modeling time MARS object (cannot point to connections)
- *Set*: container through a conditionalization association with multiple MARSC objects
- *Connection*: directional binary association visualized as a (possibly unidirectional) line
- *Attribute*: gives types values to any MARSC object
- *Aspect*: describes visualization rules (filters) for types of objects.

Table 1 shows how the generic modeling concepts are mapped into GME types. Armed with a deep understanding of the purpose/usage of abstract modeling concepts, and a sufficient understanding of the mapping of those concepts into the GME framework, it is possible to build a modeling environment for any well-defined domain (or at least a representation of the domain-concepts in a sub-optimal editor for some domains). The person who designs this domain

we call the DSME designer, or the *metamodeler*; when a student successfully passes the MIC course, then they are a domain expert in the domain of metamodeling.

3. KEY METHODOLOGIES/CONCEPTS

3.1 Knowledge of domain expert

A DSME is most useful to an expert in that particular domain. Since one of the claims of domain-specific modeling is that it increases productivity through an increased level of abstraction, a useful metric of a DSME is how efficiently it can be used “out of the box” by a domain expert.

Ideally, a DSME is designed so that a domain-expert does not need a manual to perform the basic tasks of modeling, but that since the expert has a deep understanding of semantics of the domain concepts, then there should be some logical correlation between the language and the actual semantics of the interface to the DSME. For example, an expert in UML should be able to immediately begin using a class diagram DSME, which should have intuitively obvious methods to create inheritance relationships, containment associations, and create packages and abstract classes. The intuitiveness of the DSME will be directly related to how the domain concepts are encoded into abstract modeling concepts and mapped onto concrete modeling concepts of the modeling environment used.

3.2 Encoding domain concepts

Usually domain concepts are apparent based on the nomenclature, physical existence, and/or body of research for a particular domain. Once the optimal¹ set of domain concepts has been enumerated it is the job of the metamodeler to encode those domain concepts into a language.

The choice of the appropriate modeling concept for each domain concept is key to successfully encoding the domain concepts. Domain concepts that show properties of tree-like containment should utilize hierarchical structures. Domain concepts that show properties of graph-like interconnectedness should utilize (typed) associations between concepts. The method of entry for the domain user should be efficient, as well. For example, rather than have hundreds of types of similar objects from which to choose, it is more efficient to have one type with an enumerated list of attributes to choose the specifics for this type. This reduces the set from which a domain expert may choose domain concepts, providing a less cluttered beginning point (recall that we want a domain expert to use this environment without a manual).

These examples of canonical abstraction methods are important, but are not so different from normal software abstraction when creating a class hierarchy and deciding whether to associate objects by containment, or by reference. What makes encoding domain concepts into a modeling language more interesting—and more useful when done correctly—is the ability to reflect the common notations and representations in the domain. This means that while abstraction using some modeling concept may actually be less efficient than a clever abstraction, the one that “looks” more like the notation of the domain expert is preferred.

¹Here, optimal means that the domain-expert is satisfied with the level of abstraction of the domain concepts.

Finally, metamodelers (like software architects) should keep in mind the possible future evolution of the domain when creating the design. Using inheritance concepts, it is possible to create abstractions of types in the metamodel which can be extended in future generations of the language (e.g., a generic type “network adapter” which can be specialized in future generations of the language by adding new types) while relying on common interfaces as defined by the generic type.

3.3 Intelligent translators

Of course, for a DSME to be useful, it must provide more than a drawing package; there must be a way for the domain expert to configure or generate applications using the DSME. An intelligent translator has abstract knowledge of the semantics (i.e., the intrinsic behavior) of the domain concepts, and is capable of producing one or more artifacts in the application domain when executed (i.e., the emergent behavior). The current state of the art in the design and implementation of translators is to create software through traditional software engineering techniques. However, research is continuing in the automated (and model-based) design of translators. For the purposes of the MIC course, there are two basic types of translators: those based on O-O techniques, and those based on visitor design pattern techniques.

O-O based translators are built based on the premise that each domain concept has precisely one semantic interpretation. When the translator is created, a class is assigned to each type in the domain; when the translator is invoked, then the intrinsic behavior of each instance of a domain concept is executed (by instantiating the class) to produce the emergent behavior. O-O based translators are a good fit for domains that are fairly static, and which have a well-defined, singular semantics.

However, such domains are the exception rather than the rule. More frequently, a domain has more than one “interpretation”, that is, that the semantics of the domain concepts varies depending on the target application. In this case, a more appropriate design for the translator is based on the visitor pattern. This enables the existence of a base translator that accepts a generic kind of visitor class, and then derived visitors are specialized to provide specific semantics based on the desired application. This allows the use of advanced software architecture techniques such as polymorphism in the definition of the visitor classes, to reduce the amount of code that must be written to create the translator.

3.4 So what??

Possibly the *most* important question that any designer must learn to ask is “*so what??*” That is, clever designs and techniques are nice, but what do they buy you?

For example, a metamodeler cannot just say that they have created a DSME, and just leave it at that. *So what? What domain? How general-purpose is it? What restrictions are placed on the input set? What are the intended output artifacts of the interpreter for this language?* A metamodeler cannot claim to have accounted for managing documentation through the models, without being more specific. *So*

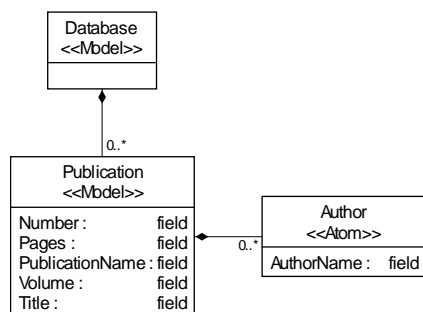


Figure 1: A prototype of the PublicationDb DSME language, after an in-class exercise.

what? How is the documentation generated? What form is the documentation in?

In addition to the justifications to provide, and questions to anticipate, students learn that they should also give their clients a sort of “so what?” questionnaire. If, for example, clients say they want to be able to model the tools in a toolbox, metamodelers should ask, *so what? What kinds of tools? What is the intended output of the modeling environment? Who will be using the DSME?* The degree of specificity of the models and generated artifact(s) are also driven by the “so what?” question. *You are generating a file—do you need to specify whether or not you use Unix or Windows carriage returns?* The “so what?” question is a guiding force in the overall design of all DSMEs, and is the first real lesson taught in the MIC course, through an example such as the one presented in the next section.

4. PUBLICATIONS DATABASE: AN EXAMPLE DOMAIN

In order to guide students through the MIC course, with its abstract ideas and (occasionally) subtle differences in concept it is *imperative* to have an example domain with which to concretely express the concepts of the course. We suggest as one such paradigm the domain of an academic publications database.

4.1 Developing the metamodel

Most graduate students are familiar with academic publications (or they will be soon enough). A database of publications is essential when compiling the references for a publication (e.g., a BibTeX database), as well as a personal compendium of papers written by one person (for biographical/resume purposes). Often, these databases are desired in more than one format, to be displayed in an academic paper, as well as on a webpage, or in a resume.

An in-class exercise will quickly identify the major domain concepts: author, title, page numbers, publication name, volume, number, etc. A quick metamodel will show how these are mapped onto GME modeling concepts, resulting in a prototype of the DSME language in about 45 minutes of classtime. Fig. 1 shows the prototype metamodel.

Once students are permitted to play with this language and try it on a few examples, it rapidly becomes clear that the

language is lacking in several respects. First of all, each publication requires the creation of a separate atom for each Author. If an author is associated with more than one publication, then there is no way to reuse the existing definition for that author, except to copy the atom into the new publication. This, however, has its problems if author information is found to be in error, since each atom would have to be modified (a daunting task for prolific authors!). Another inadequacy of the language is that there is more than one type of publication, yet only one basic type (Publication) is present.

After using the prototype, students begin to understand how the language can be *more* or *less* domain-specific, with subtle changes to the language. Should one massive type of publication be defined, that can have the full set of attributes for any conceivable kind of publication (and therefore trusting the domain expert to know which values are valid for which kind)? Or should there be a different type for each kind of publication, totally customizing the set of all publications to avoid ambiguous entries for publications (e.g., specifying a volume/number value for a publication such as a book, that probably has no such value)?

Answering these questions (and others like them) guides the development of the domain-specific language, and students inevitably decide that a more customized solution (with restricted attribute sets for publications such as books and magazine articles) is preferred. In addition, students also decide to change the way that Authors are contained in a Publication, preferring to use the Reference GME concept to define a database of Persons, and contain as a reference the Author(s) of a Publication. A more sophisticated metamodel of this language is shown in Fig. 2, showing the new addition of Editor and Person to the kinds of people, and defining that only Books can contain editors, whereas both Books and MagazineArticles can contain Authors (note also that a Person is never associated with a Publication, only when playing the role of Author/Editor).

4.2 Taking advantage of domain experts

When continuing to along the path of customizing this domain, students take one of two different roads: either they immediately begin blindly specializing publications into dozens of possible types (eventually re-examining their efforts in dismay), or they stall, faced with the prospect of defining the classifications for all publications in order to solve their problem. At this point the instructor is faced with a choice: either let the students wallow in dismay and wait for them to come the following realization themselves, or let them in on the secret. The secret, of course, is that they should be asking the “so what?” question right about now.

The purpose of this domain is to allow researchers/authors to create database artifacts for use with publishing and biographical tools. The main way in which the tool will be used is to generate a BibTeX database of selected publications (i.e., a portion of—not the entire—database). Also, generation of HTML formatted text would enable a modeler to generate listings of items published by themselves (or another person). This means that the language should not be a model of BibTeX, but instead should be a model of *publications*; however, it is true that the BibTeX specification

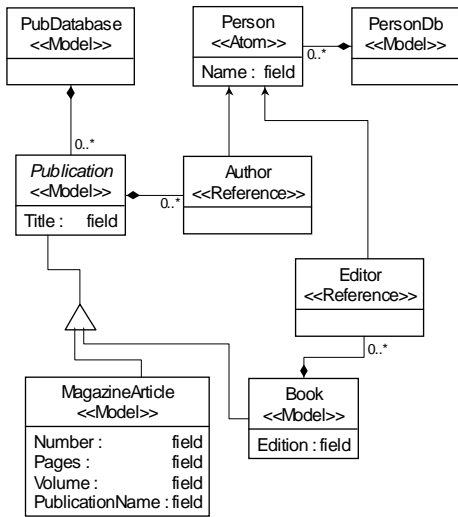


Figure 2: The revised version of the PublicationDb metamodel. Note the more mature expression of Author/Editor, and specialization of Book/MagazineArticle.

is an authoritative (and accepted) categorization of many kinds of publication classifications. Therefore, it is possible to take advantage of the work that hundreds of researchers have put in to the BibTeX domain to define the full set of classifications, while at the same time defining the PublicationDb language to be somewhat independent of BibTeX as an input format. By studying the BibTeX definitions, students begin to understand that adopting the notation and semantics of the existing domains is an effective process for designing a DSME.

Continuing in the “so what?” thread, students continue to build prototypes of the DSME, and experiment with the language. Thanks to GME’s rapid design turnaround, it’s possible to play with several dozen prototypes in the space of a few hours (limited by the imagination and keyboard skills of the modeler). A subsequent DSME metamodel is shown in Fig. 3; note that the major BibTeX types are present, with some additions and modifications. The term JournalPub is used rather than Article, and additional types of Conference and Workshop are included, to reflect these additional specialities. These types are created for categorizing publications types for biographical, more than academic, reasons.

Another enhancement in this newest version of the PublicationDb DSME is that many Databases can exist, and that a Publication can be defined once, but referred to in multiple Databases. This allows quick usage of papers that are frequently referred to, and a possible categorization of Databases as deemed appropriate by the user (e.g., the set of all publications written in 1999, or all mathematical publications) and swift reference in individual databases for generation of BibTeX files. In this case, types of publications are contained in special models that use the concept of information hiding to hierarchically separate different kinds of publications. An example definition of these enhancements

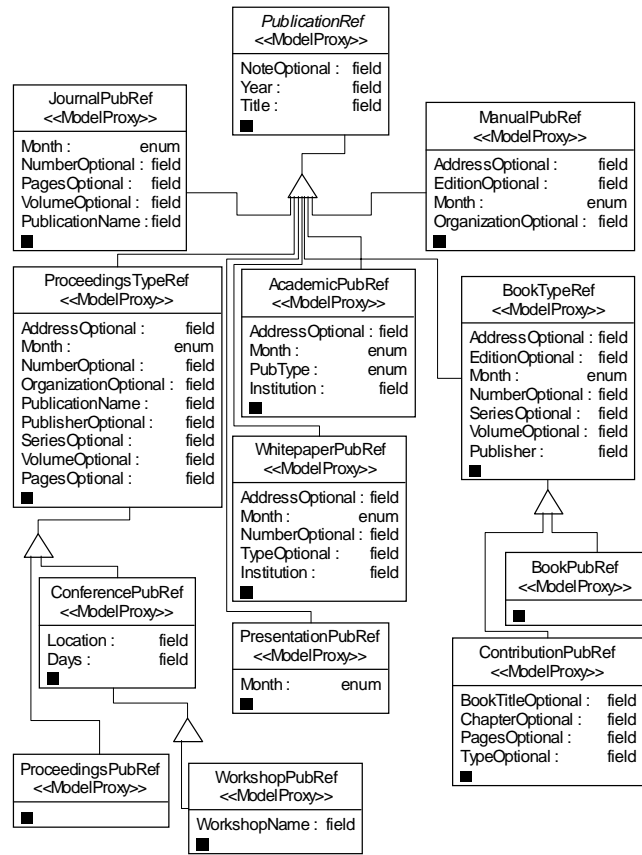


Figure 3: Excerpt of metamodel after taking into account the domain knowledge of BibTeX publication definitions. The black boxes on the classes indicate that the class is more fully defined elsewhere in the diagram.

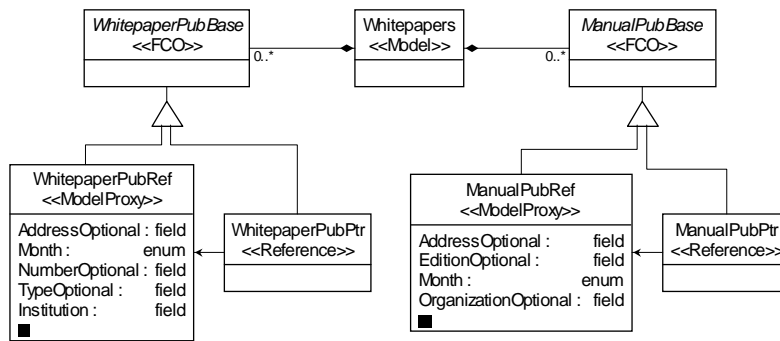


Figure 4: Enhancement to the DSME to allow references to Publications (for rapid reuse at modeling time). Note also that the types of ManualPub and WhitepaperPub are grouped into the Whitepapers container, in order to categorize them at modeling time.

for unpublished whitepapers is shown in Fig. 4.

4.3 Translators

Sooner or later, the DSME language becomes finalized, and the time comes to create the intelligent translators that convert from the domain-specific models to the artifacts. Students should have been thinking about what the artifact would look like throughout modeling time, but inevitably some have not done this. When beginning to write their first translator, a student will usually make modifications to the metamodel several times before being pleased with their translator. This is because they did not realize (at metamodeling time) how certain design patterns would enable them to take advantage of O-O software construction methodologies, such as inheritance.

GME provides a high-level C++ interface to a modeling environment’s objects, allowing a programmer to interact with object databases as objects in their program. The types defined in a metamodel can be mapped onto C++ classes, providing the ability to use software architecture methods, such as virtual methods, inheritance, get/set functions for parent/child relationships, etc. By creating abstract base types in the GME metamodel, software class definitions can be absorbed into the base classes, reducing code duplication. Students realize this benefit after they begin work, and take advantage of GME’s code generator [6] to create skeleton classes that provide the domain-specific typed high-level C++ interface to manage their translator definition. A common modification in the PublicationDb paradigm is the addition of a base type PubPtr that can point to an abstract Publication, rather than a specific derived type. This definition is shown in Fig. 5.

Students now begin to realize the relative strengths of O-O and Visitor-based translators. By creating a classic O-O definition of the classes, it is possible for commonly used methods (or “executions”) of the class types to be created. However, an HTML translator does very different things to the PublicationDb paradigm than does a BibTeX generator; thus, to create two different class hierarchies, each with a different behavior (but common underlying information gathering) is quite inefficient, resulting in code duplication. Combining the two into one class definition is also a poor

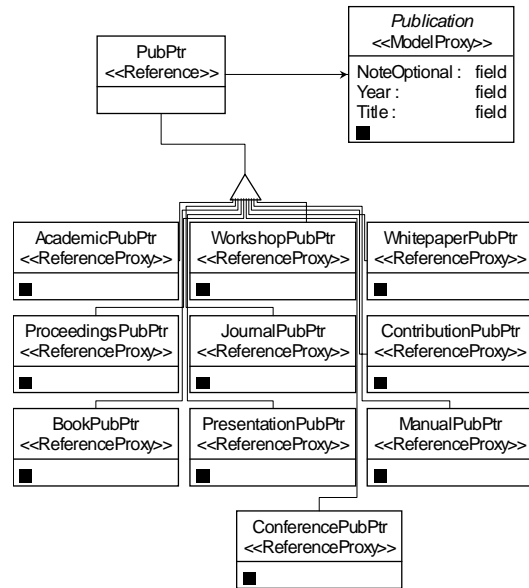


Figure 5: Metamodel enhancement usually performed after the translator writing has begun, in order to take advantage of inheritance structures in the software definitions.

design, since this requires modifying the class definitions to add any future translators.

Armed with this foresight, students create two different visitor-based translators: one that uses the O-O definitions contained in the generated C++ interface to gather BibTeX information, the other that uses the O-O definitions to gather HTML information. This data is formatted quite differently for each visitor, although the gathering is quite similar. This justifies two different designs, each of which actually has a different *execution semantics* for the PublicationDb paradigm.

5. CONCLUSIONS

As students perfect their PublicationDb design, they do eventually find an answer to the “so what??” question. The answer to this question is different for each student. Some find that they prefer more design time for the DSME/translator to have a “slicker” product, while others prefer a “beta-software” version that requires tender-loving-care from the domain expert to coax properly generated files. Since each student is the domain expert of their DSME, it is not surprising that the final designs are developed to different levels of maturity; however, each student is also responsible for *justifying* their design to the instructor, meaning that there must be a satisfactory answer to “so what??” when it comes from the instructor—and that the instructor (being also a domain expert in publications) should be able to use the final DSME without a users manual, to at least some degree of competency.

The paradigm of PublicationDb is a fascinating example of how common domains can be abstracted into formal modeling languages. It is also a good example of how agile programming can be applied through the MIC toolsuite. In fact, working prototype versions of the PublicationDb DSME can be created in as little as 4 hours (by a DSME design expert, such as one of our students), or several weeks of clever design principles and skilled development of the DSME can yield a mature, highly advanced version of the DSME which is suitable for public release.

The reason that this paradigm is so useful for instructing students in MIC design principles is that it is quickly well-understood, allows deep understanding of abstract, as well as GME-specific, modeling concepts through example, and can grow in complexity throughout the term—requiring the students to eventually ask “so what??”

If you find that you yourself are asking “so what??” to the PublicationDb DSME, you can download it for experimentation (after installing GME4 from the Vanderbilt Website) and play with the design environment for archiving (and publishing) your own bibliographic, and biographical, databases. Eventually, you will find some feature you desire, or some use case that was unaccounted for, and you may ask yourself whether it is worth implementing, or if it is outside the bounds of the DSME as used by most domain experts. In fact, the authors would be disappointed if several such suggestions were made to us to increase the usability of the tool in such a way. We would, of course, reply that if we made the changes, then “so what??”

The tool is available from <http://www.eecs.berkeley.edu/~sprinkle/oops1a/>. The BibTeX database used in the production of the document was created using the same version available from the website.

6. REFERENCES

- [1] J. Gray, S. Neema, T. Bapty, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Comm. of the ACM*, pages 87–93, Oct. 2001.
- [2] J. Gray, J. Sztipanovits, D. Schmidt, T. Bapty, S. Neema, and A. Gokhale. Two-level aspect weaving to support evolution of model-driven synthesis. In R. Filman, T. Elrad, M. Aksit, and S. Clarke, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
- [3] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), Dec. 2000.
- [4] A. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, pages 44–51, Nov. 2001.
- [5] A. Lédeczi, G. Nordstrom, G. Karsai, P. Völgyesi, and M. Maroti. On metamodel composition. In *IEEE CCA 2001*, pages CD-ROM, Sept. 2001.
- [6] S. Nordstrom, S. Shetty, K. G. Chhokra, J. Sprinkle, B. Eames, and A. Lédeczi. Anemic: Automatic interface enabler for model integrated computing. In *Generative Programming and Component Engineering (GPCE '03)*, Sept. 2003.
- [7] J. Sprinkle. Model-integrated computing. *IEEE Potentials*, 23(1):28–30, Feb. 2004.
- [8] J. Sprinkle, G. Karsai, A. Lédeczi, and G. Nordstrom. The new metamodeling generation. In *Proceedings of the IEEE Engineering of Computer Based Systems*, page 275, Apr. 2001.
- [9] J.-P. Tolvanen, J. Gray, and S. Kelly, editors. *ACM OOPSLA Workshop on Domain-Specific Visual Languages*, ISBN: 951-39-1056-3/ISSN: 0359-8470, Oct. 2001. University Printing House, University of Jyväskylä, Finland.
- [10] J.-P. Tolvanen, J. Gray, and M. Rossi, editors. *Second ACM OOPSLA Workshop on Domain-Specific Visual Languages*, ISBN: 951-791-726-0/ISSN: 1235-5674, Nov. 2002. University Printing House, University of Jyväskylä, Finland.
- [11] J.-P. Tolvanen, J. Gray, and M. Rossi, editors. *ACM OOPSLA Workshop on Domain-Specific Modeling*, ISBN: 951-39-1582-4/ISSN: 1239-291X, Oct. 2003. University Printing House, University of Jyväskylä, Finland.