Specifying Graphical Modeling Systems Using Constraint-based Metamodels

Gabor Karsai, Greg Nordstrom, Akos Ledeczi, Janos Sztipanovits
Institute for Software Integrated Systems
Vanderbilt University
230 Appleton Place, Suite 248
Nashville, TN 37203
Email: (gabor, gnordstr, akos, sztipaj)@vuse.vanderbilt.edu

Abstract

Embedded computer-based systems are becoming highly complex and difficult to implement due to the large number of concerns designers must address. These systems are tightly coupled to their environments, requiring an integrated view that encompasses both the information system and its physical surroundings. Mathematical analysis of such systems necessitates formal modeling of both "sides," including their interaction. There exist a number of suitable modeling techniques for describing the information system component and the physical environment, but the best choice changes from domain to domain. We propose a two-level approach to modeling that introduces a meta-level representation. Meta-level models define modeling languages, but they can also be used to capture subtle interactions between domain level models. We show how the two-level approach can be supported with computational tools, and what kinds of novel capabilities are offered.

1. Introduction

Effective and efficient design of control systems has challenged engineers since ancient times, when experience and intuition were the primary design tools. It was not until the late 19th century that intuition was augmented with mathematic formalism. Modern control theory can be traced to J.C. Maxwell's seminal stability analysis of the flyball governor found on Watt's steam engine, which resulted in the concept of a control system's *characteristic equation* [1]. In the early- to mid-20th century, large advances in control theory were driven by the need to control artillery in both world wars [2]. The latter part of the 20th century was dominated by advances digital control techniques, incorporating digital computers as active control elements.

Of course, the digital computer has affected all domains of engineering. Computer-based systems (CBSs), where functional, performance, and reliability requirements demand the tight integration of physical processes and information processing, are among the most significant technological developments of the past 20 years [3]. CBSs operate in ever-changing environments, where changes in mission requirements, personnel, hardware, support systems, etc., all drive changes to the CBS. Rapid reconfiguration via software has long been seen as a potential means to effect rapid change in such systems.

A CBS is essentially a control system that consists of an information processing (IP) component, a physical environment (PE), and sensing and actuation mechanisms establishing an interface between the two (Figure 1). The behavior of the resulting system is determined by all the components in this ensemble: the hardware and the software of the IP component, the interfaces to the physical processes, the physical environment, and the interaction among all of these. We argue that to develop the engineering science of these systems one needs an integrated approach, where all aspects of the design can be analyzed.

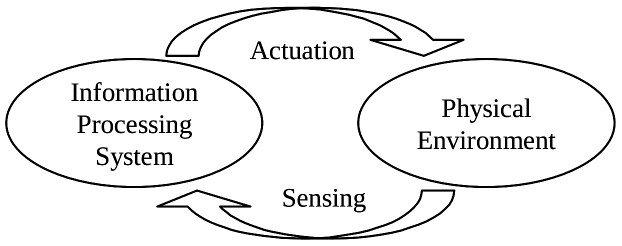

Figure 1: A Computer-based System

In any engineering discipline the rigorous analysis of a design artifact happens through the manipulation and analysis of mathematical objects, called *models*. Frequently physical prototypes are also built for experimentation, but still the analysis—and the understanding—happens with the help of mathematical objects. We need a similar model-based approach to CBS. These models, by the very nature of the CBS, must be able to represent both the IP and PE components, as well as the interaction between the two.

An illustrative example can be found in the area of digital avionics. Let us consider a fly-by-wire system that transforms pilot commands and data from environmental inputs (e.g. from air data computers and motion sensors) into actuator commands that act on the aircraft's control surfaces. When *designing* such a system, one uses the knowledge of control theory, aircraft dynamics, and other engineering disciplines to establish the control laws, to calculate controller gains, etc. The physical environment—aircraft body dynamics, actuator dynamics, etc.—determine how the IP component should behave. When *implementing* such a component one works with software abstractions: modules, tasks, synchronization, floating-point and fixed-point variables, task timing, jitter, etc. *The essential problem of CBS design is the subtle interaction between the IP and PE of the system.* Hardware or software implementation decisions have an impact in terms of the physical environment. For instance, selecting a particular fixed-point representation for a physical quantity determines expected maximum and minimum values of that quantity. The IP will simply not work if these assumptions are violated by the physical environment. Conversely, time constants determined by the physical environment will have an impact on the hardware and software implementation. This leads to a vicious circle of interaction, where changes on one side impact the other and vice versa. In order to understand CBS it is not sufficient to model only the IP or PE components—we need techniques for simultaneous modeling that also support capturing the interactions.

In order to analyze, validate, and predict the behavior of the integrated system from such models, the modeling language should be rich enough to capture all these aspects. Additionally, if feasible, we desire to *synthesize* (automatically generate) the implementation of the system from the models and component libraries. This is made possible by the development of various design automation algorithms and tools. Design automation is very successful in the hardware world but only recently have software synthesis tools begun to emerge.

In this paper, we address the following questions: What is the right way to model CBS? What is the "modeling language" to be used? *We argue that there is no single modeling language which satisfies the requirements of all CBS.* Instead, we propose a two-level approach, where area-specific modeling tools are used for creating domain-specific models, and these tools are represented in terms of (and built from) a higher-level *metamodel*.

2. The Vision

In designing CBS hardware and software, one must use *domain-specific* terminology, concepts, and techniques. By domain, we mean the larger engineering discipline within which the CBS exists. CBS are often the result of cooperation between domain engineers and hardware and software designers. We argue that the common language used by these participants should be that of the *domain* and not necessarily that of computer engineering.

Modeling languages that capture interesting properties of software systems (e.g. UML) are rarely suitable for modeling an entire system. Note that the "entire system" includes not only the hardware and the software, but the environment as well. While there are some aspects of UML that make it suitable for modeling dynamic, reactive systems (e.g. state charts), it is inadequate for capturing models in the form of Laplace transforms or differential equations. Mature engineering disciplines (e.g. control theory or chemical engineering) have their own languages—forcing the use of another modeling language is not acceptable.

Another aspect of CBS is their integrated nature. They integrate different disciplines: hardware design, software engineering, performance modeling and engineering, in addition to the "base" domain engineering discipline. When one creates models for such systems, it is necessary that the models be integrated. For example, models of the software architecture should be considered in conjunction with models of the hardware system to determine end-to-end timing latencies. Therefore, while an engineering modeling language dominates the modeling process, one must also address the issue of integrating these models with models that are closer to the domain of computer engineering. We argue that integration of models is not only an opportunity but a necessity for any kind of analysis and synthesis of complex CBS.

3. The Solution

The vision presented above seems to introduce significant difficulties. We know we need domain-specific modeling approaches. We also need to integrate models of differing disciplines. Both of these goals can be achieved by using appropriate tools, but at a very high cost—the development of customized modeling and integration solutions is very expensive. To counter this, we present an approach that is based on introducing a second level of modeling, called the *meta*-level.

We propose to use a higher-level, meta-level modeling language. The meta-language is not used for defining domain *models*, but rather for defining domain-modeling *languages*. Thus, "sentences" in the meta-language define specific domain languages, while "sentences" of the domain language define specific systems.

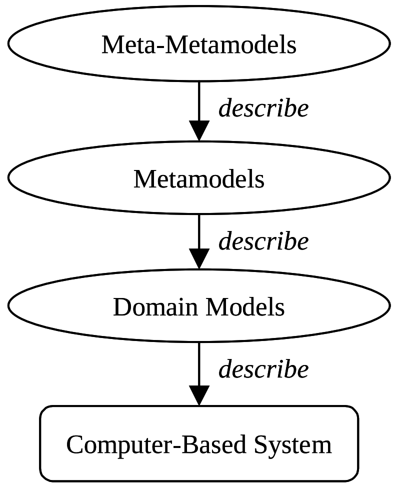

Figure 2: The four layers of modeling

Figure 2 shows the four layers of modeling that one can achieve using this approach. The real CBS is described in the form of various domain models. Metamodels describe how domain models are organized: their ontology, syntax and semantics; i.e. the *language* used to define domain models. Additionally, meta-metamodels define how metamodels are organized, their ontology, syntax and semantics; i.e. the *language* used to define metamodels. The key to this approach is that a lower layer is always described in terms of the constructs of the higher layer.

Using the metamodel one creates a *domain specific formal modeling language* that is then used to create domain models of the actual system. Formally, a modeling language can be defined as a triplet of ontology, syntax, and interpretation:

$$L = <O, S, I>$$

The ontology defines the concepts and their relationships in the language, the syntax defines all the (syntactically) correct sentences of the language, and the interpretation defines the semantics: the meaning of those correct sentences. A domain-specific modeling language consists of domain-specific ontology, syntax, and interpretation:

$$L_D = <O_D, S_D, I_D>$$

The domain models—the syntactically and semantically correct sentences of L_D built from instances of concepts and relationships defined in the domain ontology O_D—represent the CBS: its IP and PE components, along with the interactions among them. A meta-level language

$$L_M = <O_M, S_M, I_M>$$

consists of the ontology for defining domain-level languages, the correct syntax of those domain-language definitions, and their interpretation. The metamodels—syntactically and semantically correct sentences of L_M built from instances of concepts and relationships defined in the meta-ontology O_M—define L_D in terms of $<O_M, S_M, I_M>$. This implies that a meta-language must allow us to define *ontologies*, *syntax*, and *interpretation* in a mathematically precise way.

Having an explicit meta-specification of the domain modeling languages also helps when integrating models of different domains. On the meta-level one can express the relationships and dependencies among different domain-specific concepts, thus specifying the *rules* for combining different domain models. Formally, a metamodel may define more than one L_D, and may include $<O_{Di,j}, S_{Di,j}, I_{Di,j}>$ that captures ontology, syntax and interpretation for the crossing of domains D_i and D_j. Obviously, the explicit specification of these interdependencies can also be used to constrain the domain specific modeling language to only those constructs where the integration is meaningful.

Another important result of our approach is the ability to *evolve* the modeling tools over time in a formally verifiable manner. Just as domain experts evolve a particular CBS by updating its domain models and regenerating the CBS, the domain-specific modeling tools themselves are evolved by modifying the metamodel and regenerating the tools. Also, the "before and after" metamodels provide a framework for providing an automated domain model migration process.

To summarize, we advocate a two-step process for modeling CBS. In phase one, a domain-specific modeling language is described using a metamodeling language. We call this development the metamodel of the domain. To support reusability, metamodels of proven domain modeling approaches (e.g. finite state models, data flow models, etc.) should be available in a metamodel library to allow rapid composition of metamodels. In phase two, the domain-specific modeling language is used to build the models of actual systems.

4. The Implementation

While conceptually clear, the approach described above is useful only if appropriate tools are available. ISIS has been engaged in developing the supporting infrastructure for the two-level modeling approach since 1994. The detailed results of this research have been reported elsewhere [4]. Here we give a summary of the technical approach.

As mentioned earlier, the domain-level language $L_D = <O_D, S_D, I_D>$ used to specify CBS models is defined using concepts provided by the ontology component O_M of the meta-level language $L_M = <O_M, S_M, I_M>$. Below we describe the capabilities of the components of L_M.

4.1 Metamodel ontology: O_M

A metamodeling language must allow the definition of the modeling concepts used to define systems within the domain. Modeling concepts include not only the actual concepts of the domain (e.g. data streams and stores, processes, dataflow networks), but also *standard modeling abstractions*—patterns that provide a prototypical solution to a modeling problem—directly supported by the tools. Many such modeling abstractions exist in engineering but are often focused on a particular solution space or sub-domain. We claim that a core set of fundamental modeling abstractions exists and they are largely adequate to express the design concepts, notions, and artifacts used across engineering domains. Table 1 below lists the elements of this set.

We have chosen a metamodeling approach where some of these abstractions are first-class concepts (i.e. they can be instantiated), while the remaining abstractions are supported through special embellishments on the basic metamodeling constructs.

4.2 Metamodel syntax: S_M

Our metamodel syntax is essentially the same as that of UML class diagrams [5] and OCL expressions [6]. Additional, non-UML syntactical constructs are used for two purposes: (1) to indicate the use of other fundamental modeling abstractions (e.g. module interconnection and multiple aspects), and (2) to control how domain models are to be visualized. Their specific capabilities and concrete syntax is discussed elsewhere [8].

Table 1: Fundamental modeling abstractions

Classes	Specific classes of entities that exist in a given system or domain. Domain models are entities themselves and may contain other entities. Entities are instances of classes. Classes (thus entities) may have attributes.
Associations	Binary and n-ary associations among classes (and entities).
Specialization	Binary association among classes with inheritance semantics.
Hierarchy	Binary association among classes with "aggregation through containment" semantics. Performs encapsulation and information hiding.
Module inter-connection	A specific pattern of relationships among classes. Classes can be associated with each other by connecting their ports (specially marked atomic entities contained in the classes).
Constraints	A binary expression that defines the static semantic correctness of a region of the model: if the objects of the region are "correct," the expression evaluates to "TRUE."
Multiple aspects	Allows partitioning a complex model according to part categories. Used for visibility control, but may also be used for aggregating specific properties of models with respect to specific concerns.

4.3 Metamodel construction and semantics

We have created a metamodeling tool that supports the visual construction of metamodels [9]. The metamodeler uses this tool to first construct the core metamodel using UML class diagrams and then embellishes it with special "markers" to specify other properties of the domain modeling language that couldn't be expressed using the class diagram. Additionally, the metamodeler specifies OCL constraints that capture assertions that must be true for the domain models to be semantically correct.

The meaning (i.e. the semantics) of a metamodel is defined through a domain-modeling tool. We use the following pragmatic definition for the semantics of a metamodel: *A metamodel is a program that, when "executed," configures a generic modeling environment (GME) to support a domain-specific modeling language. The domain-specialized instance of the GME allows only the creation of syntactically and semantically correct domain models, as defined by the metamodel.* This concept is illustrated in Figure 3 below. Interestingly, this principle and approach makes possible a very high degree of reuse in the modeling tools. In fact, we are using the _same_ GME as the foundation tool for metamodeling and domain modeling. We have a meta-metamodel that configures the environment to support metamodeling. Thus, we can extend our metamodeling language, although this typically necessitates changes in the GME as well.

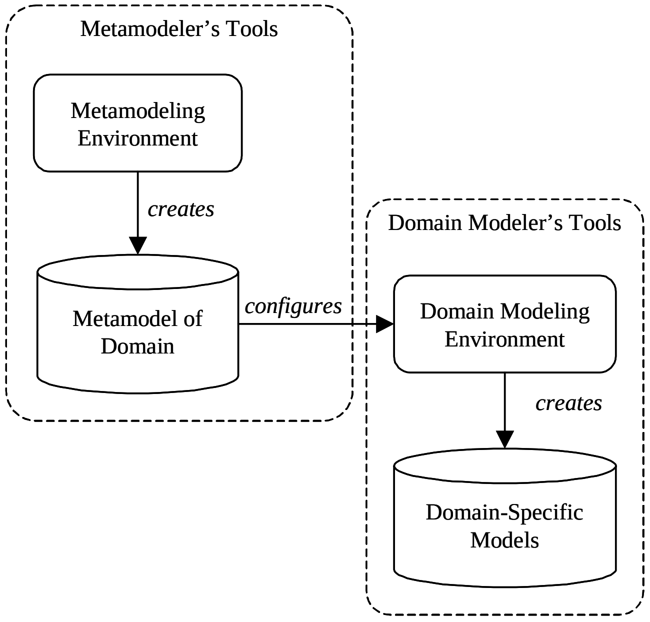

Figure 3: Metamodeling and Domain Modeling

It is worthwhile to see how metamodel concepts map into the specific capabilities of the domain-modeling environment. Embellished UML classes are turned into *atoms* (primitive, iconic components of a drawing that have no structure, only attributes), *models* (complex diagrammatic constructs that have structure and attributes, and contain atoms, models, and connections), and *connections* (attributed connectors on the diagrams that relate precisely two atoms or models). The metamodel specifies the composability constraints on these objects. If a metamodel class embellished as a "model" aggregates another class embellished as an "atom," that means that the domain models may contain atoms of that type. This semantics is enforced in the domain-modeling environment when the user attempts to add an atom to a model. Connections are derived from associations on the UML diagram: a connector is legal between any two domain objects (model or atom) whose original classes in the UML class diagram are connected (i.e. associated). Further details of the interpretation of the UML class diagram as a configurator for domain-modeling can be found in [8]. There are a few other modeling constructs not discussed here. The interested reader is referred to the detailed documentation of our tools [10].

The OCL constraints specified in metamodels are checked at domain model construction time. When a constraint evaluates to FALSE, it indicates that the current model violates the static semantics of the domain modeling language. This technique is best illustrated by a simple example. Consider the following metamodel (Figure 4) of a `Hose`, where the attribute `threadSize` is used to model the size of the male and female connectors at the ends of the `Hose.` A `Hose` can be connected to other `Hose`s to form a chain via `HoseConnection`s. Obviously, the connection has a source and a destination `Hose`.

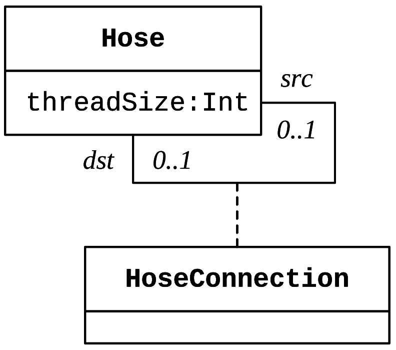

Figure 4: Metamodel of Hoses and HoseConnections

When connected together, each end of a `Hose` plays the role of `src` or `dst`. Since the multiplicity of each association end is zero or one, this implies that each end of a `Hose` can connect to at most one other `Hose`. Let us assume that we have two additional constraints when connecting `Hoses` together. First, both `Hose`s must have the same `threadSize`, and second, a `Hose` should not be connected to itself. Note that neither of these constraints can be stated using only UML class diagrams. We must specify these constraints using OCL, as shown below:

```
HoseConnection.allInstances->
                  forAll(c|c.src.threadSize
                  = c.dst.threadSize)
HoseConnection.allInstances->
                  forAll(c|c.src <> c.dst)
```

When a domain model is edited, these expressions are evaluated, and an error is signaled when they fail.

4.4 Metamodel- and domain-model semantics: I_M and I_D

The semantics of a metamodel as discussed above is limited to an interpretation in the context of the GME. This allows us to build syntactically and semantically correct domain-specific models, but not much else. We want to build a system from the domain models and determine properties of that system via various engineering tools. The domain models play a crucial role, as they are the subject of (or input to) various analysis and synthesis procedures. *These procedures assign a dynamic semantics to the domain models.*

Specifically, the dynamic, or operational, semantics of a domain model is determined in two steps in our system [4][11]. We assume that an execution platform is available, which has an "instruction set" with clearly defined semantics. The platform can be an analysis engine (e.g. a simulator package), an execution environment (e.g. a real-time operating system), or any other operational computational system. In step one, the domain models are processed by a software component called a *model interpreter* that transform the models into the "instruction set" of the execution platform. In the second step, the execution platform executes those "instructions." Thus, the domain model semantics, I_D, is realized by a transformation engine and an execution engine, as shown in Figure 5.

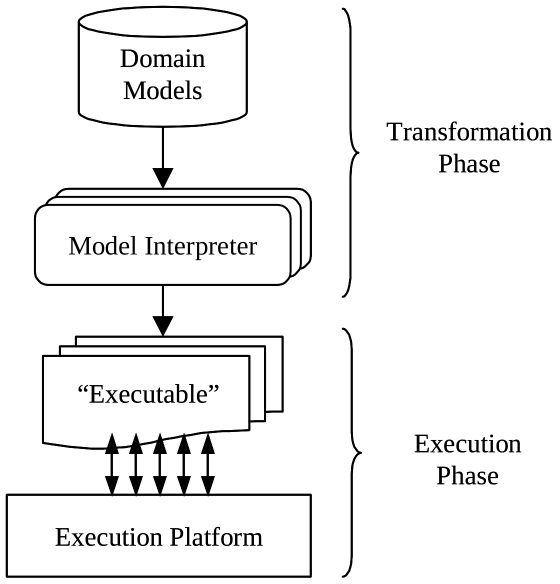

Figure 5: Assigning semantics to domain models

It seems natural that the semantics of domain models should also be captured in the metamodel of the domain. That is, I_M specifies how to map a particular metamodel into a specific I_D that determines exactly how a model interpreter works and how the execution platform processes the result of the transformation phase. As discussed above, the metamodel should not only specify the ontology, syntax, and static semantics of the domain models, but also their interpretation—their dynamic semantics. In our approach, the latter involves the formal specification of the execution platform and that of the transformation of domain models into the "instruction set" of the execution platform.

Currently, we are conducting research activities to address the formal specification of the dynamic semantics of domain models. The above two-phase scheme has been applied in many applications, by hand-crafting the model interpreters for specific execution platforms. However, this is a difficult and error-prone process. Developing a formal language for capturing the properties of the model transformation and the execution platform, and developing the semantics of that language, will allow us to speed up the development of domain-specific modeling languages and make their interpretation mathematically precise. Some of our preliminary work on the theoretical foundations of formalizing these specifications can be found in [12].

5. Comparison with Other Approaches

Many concepts and techniques in our approach are based on groundwork done by a large community of modeling experts, academic and industrial researchers. The use of metamodels for defining modeling concepts and domains can be found in many proposed engineering standards. For example, CDIF [13] proposes the use of the four-layer modeling approach. The static semantics of UML is specified using a similar approach, using UML as its own metalanguage [7]. Metamodeling is an idea that has been addressed in many research workshops and projects (e.g. [14] and [15]). Some of the relevant research activities and industry efforts are related to integrating data from various sources (e.g. MetaData coalition [16]) as well as creating domain-oriented tools for building original types of models (e.g. DOME [17]).

In comparison, our effort has focused on developing meta-level tools—modeling techniques, modeling environments, metamodel interpreters, etc.— that associate a highly pragmatic and operational semantics to the metamodel. Furthermore, our research is addressing the specific needs of CBS, where domain-specific modeling languages are often given, and we have to integrate them with other domain-specific modeling approaches.

6. Conclusions and Future Work

The two-level approach to the specification of domain-specific modeling languages (DSML) and modeling environment generation has several advantages. By specifying the entities, relationships, attributes, and constraints at the metamodeling level, the DSML can be described with mathematical precision, can be safely evolved over time, and can be used to configure a GME for use in designing CBS within a particular domain. Once configured, the domain modeling environment ensures valid model creation through the use of constraint specifications obtained from the metamodel, enforcing the formal static semantics of the domain at model editing time. Also, if the metamodel captures the mapping of domain models into the “instruction-set” of an execution platform, a transformation engine can be synthesized to facilitate that mapping.

Using the metamodeling technology described in this paper domain-specific modeling tools have been created, and have been in constant use for many years, in many engineering application areas and domains [18]. Our current work focuses on more efficient methods for mapping the abstract syntax of a metamodel onto the graphical idioms of the GME.

7. Acknowledgements

The DARPA/ITO EDCS program (F30602-96-2-0227) and the DARPA/ITO SEC program (F33615-99-C-3611) has supported, in part, the activities described in this paper.

References

[1] J. C. Maxwell, "On Governors," *Proc. Royal Soc. London*, vol. 16, pp. 270-283, 1868.

[2] N. Wiener, *Cybernetics: or Control and Communication in the Animal and the Machine*, Cambridge: MIT Press, 1948.

[3] J. Sztipanovits, “Engineering of Computer-Based Systems: An Emerging Discipline,” *Proceedings of the IEEE ECBS’98 Conference*, 1998.

[4] J. Sztipanovits, G. Karsai, “Model-Integrated Computing,” IEEE Computer, pp. 110-112, April, 1997.

[5] UML Summary, ver. 1.0.1, Rational Software Corporation, March, 1997

[6] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997.

[7] UML Semantics, ver. 1.1, Rational Software Corporation, et al., September 1997.

[8] Nordstrom G.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Ph.D. Dissertation, Vanderbilt University, 1999.

[9] A. Ledeczi, et al., “Metaprogrammable Toolkit for Model-Integrated Computing," Proceedings of the IEEE ECBS’99 Conference, 1999.

[10] Generic Modeling Environment documents. http://www.isis.vanderbilt.edu/projects/gme/Doc.html

[11] J. Sztipanovits, et al.: “MULTIGRAPH: An Architecture for Model-Integrated Computing,” Proceedings of the IEEE ICECCS’95, pp. 361-368, Nov. 1995.

[12] G. Karsai, et al., “Towards Specification of Program Synthesis in Model-Integrated Computing," Proceedings of the IEEE ECBS’98 Conference, 1998.

[13] CDIF Meta Model documentation. http://www.metamodel.com/cdif-metamodel.html

[14] Metamodeling in OO (OOPSLA'95 Workshop) October 15, 1995, http://saturne.info.uqam.ca/Labo_Recherche/Larc/MetamodelingWorkshop/metamodeling-agenda.html

[15] 43rd Annual Meeting of the International Society for Systems Sciences, at the Asilomar Conference Center, Pacific Grove, California, June 26 to July 2, 1999, http://www.isss.org/1999meet/sigs/sigmodel.htm

[16] MetaData coalition. http://www.mdcinfo.com/

[17] Dome Official Web Site, Honeywell, 2000, http://www.src.honeywell.com/dome/

[18] Model-Integrated Computing documents. http://www.isis.vanderbilt.edu/projects/