Journal of Universal Computer Science, vol. 6, no. 10 (2000), 1131-1144
submitted: 8/6/00, accepted: 19/9/00, appeared: 28/10/00

Towards Two-Level Formal Modeling of Computer-Based Systems

Gabor Karsai
(Vanderbilt University, Nashville TN
gabor@isis.vanderbilt.edu)

Greg Nordstrom
(Vanderbilt University, Nashville TN
gnordstr@isis.vanderbilt.edu)

Akos Ledeczi
(Vanderbilt University, Nashville TN
akos@isis.vanderbilt.edu)

Janos Sztipanovits
(Vanderbilt University, Nashville TN
sztipaj@isis.vanderbilt.edu)

Abstract: Embedded Computer-based Systems are becoming highly complex and hard to implement because of the large number of concerns the designers have to address. These systems are tightly coupled to their environments and this requires an integrated view that encompasses both the information system and its physical surroundings. Therefore, mathematical analysis of these systems necessitates formal modeling of both ”sides” and 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. In this paper, 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 will show how the two-level approach can be supported with computational tools, and what kind of novel capabilities are offered.
Category: D.2.2 Tools and Techniques

1 The Need

A Computer-based System (CBS) is essentially a *physical* system that consists of an information processing (IP) component, a physical environment (PE), and a sensing and actuation mechanism that establishes the interface between the two. Figure 1 illustrates this statement. The behavior of the resulting system is determined by all the components in this ensemble: the hardware and the software of the information processing 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.

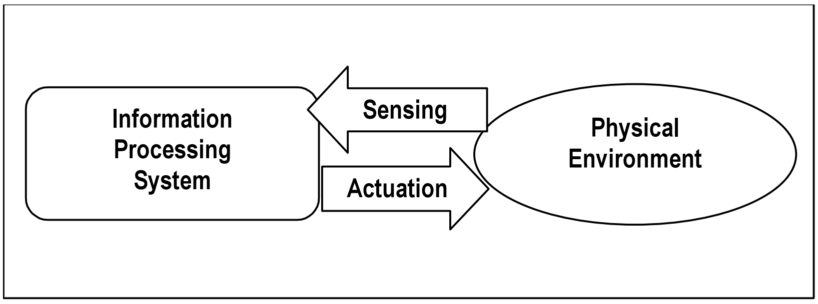

Figure 1: A Computer-based System

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 the mathematical objects. We need a similar approach to CBS, such that we can build models of the systems. These models, by the very nature of the CBS, have to be able to represent both the IP and the PE components, together with 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 control surfaces of the aircraft. 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 the controller gains, etc. The physical environment: aircraft body dynamics, actuator dynamics, etc. determine how the information processing 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 is the subtle interaction between the IP and PE of the system.* When a hardware or software implementation decision is made, that will have an impact in terms of the physical environment. For instance, selecting a particular fixed-point representation for a physical quantity determines what the expected maximum and minimum value of that quantity is. The IP will simply not work if these assumptions are violated by the physical environment. On the other hand, 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 just the IP or just the PE components, we need techniques for simultaneous modeling that also support capturing the interactions.

Naturally, we want to analyze, validate, and predict the behavior of the integrated system from these models. Hence, the modeling language should be rich enough to capture all these aspects. Additionally, if feasible, we would like to *synthesize* the implementation of the system from the model. By synthesis we mean a process that leads from design models and component libraries to automatically generated software and hardware components. This last step is made possible by the development of various design automation algorithms and tools. Design automation is very successful in the hardware world but recently software synthesis tools have also become available.

In this paper, we want to 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 would satisfy 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 *meta*-model.

2 The vision

Except for trivial cases, CBS are closely related to mature engineering disciplines. An industrial instrumentation and control system relies on signal processing, control engineering, and electronics. An on-line problem-solving environment for chemical manufacturing uses process engineering knowledge. An avionics system employs the concepts of aerodynamics, control theory, fault tolerant computing, and others. Arguably, CBS are *always* used in an engineering context, where the solution provided by the CBS must fit in.

In order to help design the hardware and the software for a CBS, one must use *domain-specific* terminology, concepts, and techniques. By domain, we mean the larger engineering discipline where the CBS belongs. 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 very rarely suitable for modeling the 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 and forcing the use of another modeling language is not acceptable.

To summarize, we emphasize the need for domain-specific modeling languages in order to properly model all aspects of CBS. As CBS are used in widely differing domains, there is a potentially very large number of modeling languages that can—and should—be used. To phrase it differently: the modeling languages for CBS changes from domain to domain.

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 these systems, it is unavoidable that these models be integrated. For example, models of the software architecture should be considered in conjunction with the 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 also 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 that we need domain-specific modeling approaches. We also need the capability to integrate models of different disciplines. Obviously, both of these goals can be achieved by using appropriate tools, but with a very high cost, because the development of modeling tools and integration solutions is very expensive. Below, we present an approach to the problem 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.

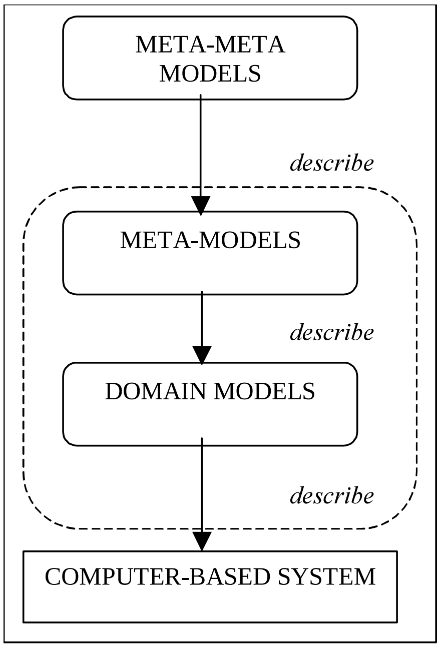

Figure 2: The four layers of modeling

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. The meta-models describe how the domain models are organized: their ontology, syntax and semantics; i.e. the *language* used to define domain models. Additionally, meta-meta models can define how meta-models are organized, their ontology, syntax and semantics; i.e. the *language* used to define meta-models. This final layer, while intellectually interesting, is rarely used in practical systems. The key to this approach is that a lower layer is always described in terms of the constructs of the higher layer.

We claim that two-level modeling, as provided by the domain models and the meta-models, is necessary to build formal models of CBS. The meta-level modeling lets us define how to build models of CBS for particular domains. Furthermore, if in one CBS a number of different (domain-) modeling languages are used, meta-level models can capture the interactions between those domains, which is crucial for integration.

The meta-level is necessary because of the variety of systems and the dissimilarities among the domains where they are applied. Using the meta-model one creates a *domain specific formal modeling language* that can then be used to create the domain models of the actual system. Once the domain modeling language is established, using its syntax and semantics one can build models of the real system. The analysis and synthesis of the real system should be done in terms of the domain modeling language, and thus the domain models.

Formally, a *modeling language* can be defined as a triplet of ontology, syntax, and interpretation:

$$L = \langle O, S, I \rangle$$

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. We claim that to specify a modeling language —its concepts, syntax, and semantics— all components of the above triplet must be precisely defined. Note that the specification of a modeling language gives a finite description of all the possible syntactically and semantically correct models one can create using the language.

We partition the set of modeling languages into two categories: domain-specific modeling languages and meta-modeling languages. Note that the distinction is slightly artificial, as a meta-modeling language is also a domain-specific modeling language, with the domain being that of modeling language specifications (i.e. meta-models).

A *domain-specific modeling language* consists of domain-specific ontology, syntax, and interpretation:

$$L_D = \langle O_D, S_D, I_D \rangle$$

The syntactically and semantically correct *sentences* of L_D built from instances of concepts and relationships defined in the domain ontology O_D, in compliance with the rules of S_D, and which also have a well-defined semantics in I_D, are the *domain models.* The domain models represent the CBS: its IP and PE components, together with the interactions among them. When one defines a domain-specific modeling language, all three elements of the triplet —ontology, syntax, and interpretation— should be precisely defined. The general observation on the relationship between the modeling language and the models (sentences) built using it still applies: the (finite) definition of a domain-specific modeling language describes a (possibly infinite) set of domain-specific models that can be expressed in the language.

A *meta-modeling language* is a modeling language that is used to define other modeling languages. Because a meta-modeling language can also be considered a domain-specific modeling language (with the domain being that of ”modeling languages”), it is desirable that the meta-modeling language be powerful enough to describe itself, in a meta-circular manner.

Formally, a meta-modeling language

$$L_M = \langle O_M, S_M, I_M \rangle$$

consists of the ontology used for defining (domain-level) modeling languages, the correct syntax of those domain-language definitions, and their interpretation. The meta-models —the syntactically and semantically correct sentences of L_M built from instances of concepts and relationships defined in the meta-ontology O_M— define any arbitrary L_D in terms of $\langle O_M, S_M, I_M \rangle$. This implies that a meta-language should allow us to define *ontologies*, *syntax*, and *interpretation*.

The relationship among meta- and domain-specific modeling languages and meta-models and domain-models is illustrated in ***Figure 3***.

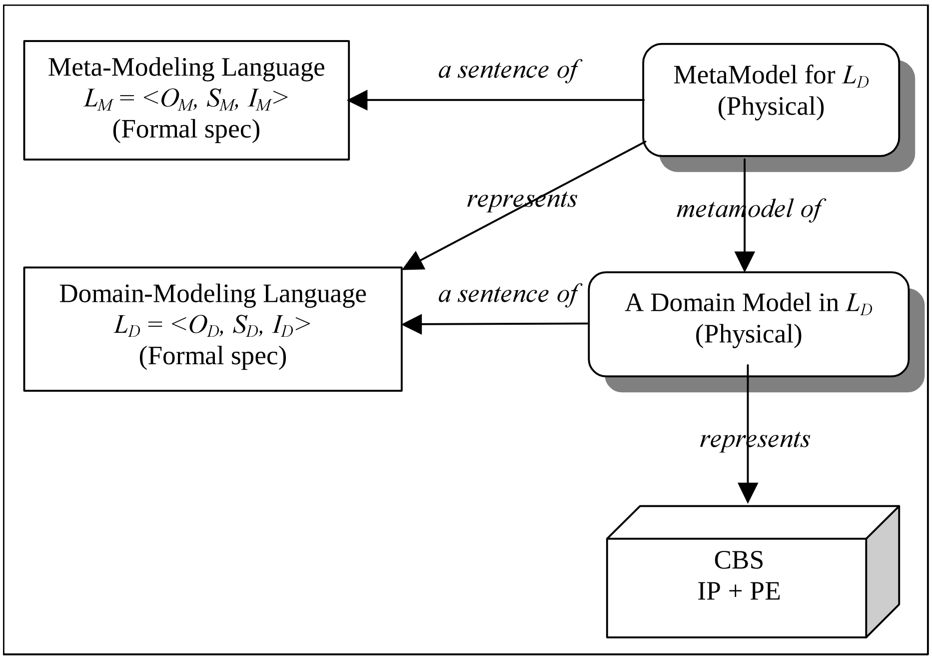

Figure 3: Relationship between meta- and domain modeling languages, and meta- and domain-models

Using this two-level modeling approach has various advantages, chiefly that the domain modeling language can be specified in a mathematically precise way. While this may seem an inconvenience, the long-term benefits of the precise definition outweigh the difficulties.

Having an explicit meta-specification of the domain modeling languages also enables in 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 meta-model $\langle O_M, S_M, I_M \rangle$ may define more than one L_D, and may include $\langle O_{Di,j}, S_{Di,j}, I_{Di,j} \rangle$ 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 outcome 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 modeling tools are evolved through modifying the metamodel and regenerating the domain-specific modeling tools. Also, by having both the pre- and post-evolution metamodels, a framework for model migration (evolution of existing models created with the ”old” modeling environment into the ”new” modeling environment) is established.

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

4 The implementation

While conceptually clear, the approach described above is useful only if appropriate tools are available. The Institute for Software-Integrated Systems at Vanderbilt University 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 [Sztipanovits 97]. Here we give a summary 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

First, 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, compound process blocks in a real-time software modeling domain), but also *standard modeling abstractions* that are directly supported by the tools. Modeling abstractions define patterns that provide a prototypical solution to a modeling problem, often related to managing complexity in the modeling process. Many such modeling abstractions exist in engineering but they are often focused on a particular solution space or sub-domain. For instance, hierarchical process flow diagrams in the real-time software modeling domain are a convenient, standard abstraction. Frequently, these modeling techniques are a domain-specialization (or superposition) of a few a core basic modeling abstractions. For instance, hierarchical process flow diagrams use the abstractions of (part-whole) ”hierarchy” and ”module interconnectivity” and apply them in the domain of processes and dataflows. Similarly, containment hierarchy is universally accepted as a method to hide or reveal detail, while inheritance is often used to represent object specialization and refinement. We claim that a core set of these fundamental modeling abstractions exists and they are largely adequate to express the design concepts, notions, and artifacts used across engineering domains. ***Table 1*** lists the elements of this set.

1. 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.
2. Associations	Binary and n-ary associations among classes (and entities).
3. Specialization	Binary association among classes with inheritance semantics.
4. Hierarchy	Binary association among classes with ”aggregation through containment” semantics. Performs encapsulation and information hiding.
5. 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).
6. 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”.
7. 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.

Table 1:Fundamental standard modeling abstractions

We have chosen a metamodeling approach where some of the above 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. In our metamodeling language, the ontology supports the construction of metamodel objects that are instances of the first four modeling abstractions. We have used the industry standard Unified Modeling Language (UML) [UML 97b] to facilitate this. For specifying constraints, we have used the Object Constraint Language (OCL) [OCL 97]. This choice was made for several reasons: UML directly supports the fundamental modeling abstractions 1-4, and OCL supports 6, as described above; the UML/OCL syntax and semantics are well defined [UML 97a]; UML/OCL is an OMG standard that enjoys widespread use in the industry. The abstractions: 1,2,3, and 4 directly map to the abstractions available in the class diagram sub-language of UML under the same name (”hierarchy” maps to ”aggregation through containment”), while constraints are supported through OCL expressions. The remaining fundamental modeling abstractions are supported through special (visual) syntactical constructs. These abstractions are directly supported by our generic modeling tool [GME 00].

To summarize, the core of our metamodeling language is the class diagram portion of UML adorned with OCL constraint equations. Our UML constructs are (visually) decorated to indicate the use of other fundamental modeling abstractions [GME 00][Nordstrom 99b].

4.2 Metamodel syntax: S_M

Our metamodel syntax is essentially the same as that of UML class diagrams and OCL expressions. Additional, non-UML syntactical constructs are used for two purposes: (1) indicate the use of other fundamental modeling abstractions (e.g. module interconnection), and (2) control how the domain model is to be visualized. Their specific capabilities and concrete syntax is discussed elsewhere [Nordstrom 99b].

4.3 Metamodel construction and semantics

We have created a metamodeling tool that supports the visual construction of metamodels [Ledeczi 99]. The metamodeler uses this tool to first construct the core metamodel (using UML class diagram) 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 can specify constraints (using OCL expressions) 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 to support a domain-specific modeling language. The domain-specialized instance of the generic modeling environment allows only the creation of syntactically and semantically correct domain models, as defined by the metamodel.* This concept is illustrated on Figure 4 below. Interestingly, this principle and approach makes possible a very high degree of reuse in the modeling tools. In fact, we are using the <u>*same*</u> generic modeling environment as the foundation tool for metamodeling and domain modeling. We have a meta-metamodel that configures the environment to support metamodeling. Thus, we can perform ”second-order” modeling as well: we can extend our metamodeling language (if so desired), although this typically necessitates changes in the generic modeling environment as well.

It is worthwhile to see how metamodeling 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). There are a few other modeling constructs: references, conditionals, etc., and the interested reader is referred to the detailed documentation of our tools [GME 00]. 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 [Nordstrom 99b].

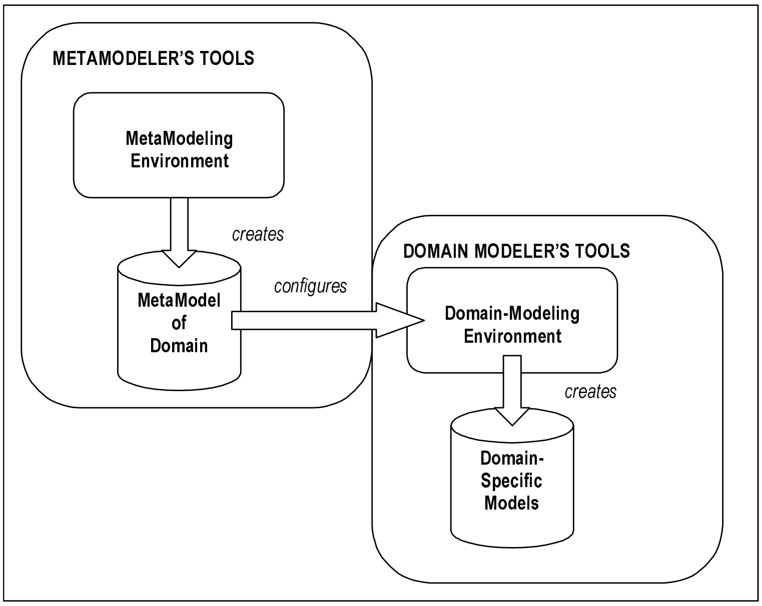

Figure 4: Metamodeling and Domain Modeling

The constraints specified in metamodels (using OCL) are checked at domain model construction time. An OCL expression is a predicate that can be evaluated in the context of the current domain model being constructed. When the predicate evaluates to FALSE, it is an indication of a violation of the static semantics of the domain modeling language by the model under construction.

This technique is best illustrated by a simple example. Consider the following metamodel (Figure 5) 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 others to form a chain via `HoseConnection`s. Obviously, the connection has a source and a destination hose.

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 on connecting `Hoses` together. First, both `Hose`s must have the same `threadSize`, and second, a `Hose` can't connect to itself. Note that neither of these constraints can be stated using only UML class diagrams. We 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 the domain modeler edits a domain model, these expressions are evaluated, and an error is signaled when they fail.

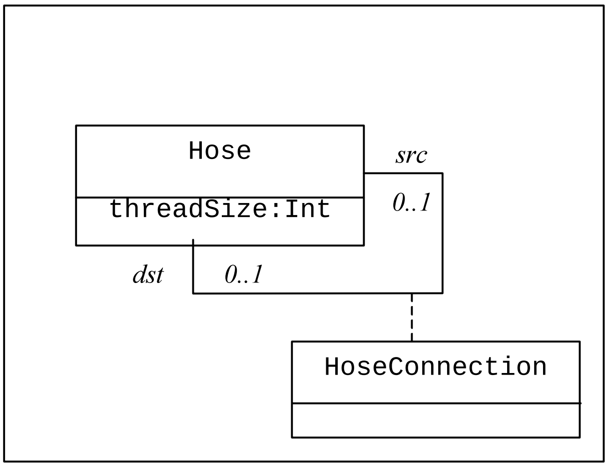

Figure 5: Metamodel of Hoses and HoseConnections

4.4 Metamodel semantics 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 generic modeling environment. This allows us build domain-specific models that are syntactically and semantically correct, 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 systems [Sztipanovits 97][Sztipanovits 95]. To begin with, 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), or 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 the *model interpreter* that transforms the models into the ”instruction set” of the execution platform. In the second step, the execution platform executes that output of the first step. Thus, the domain model semantics, $\boldsymbol{I_D}$, is realized by a transformation engine and an execution engine. This process is illustrated on Figure 6.

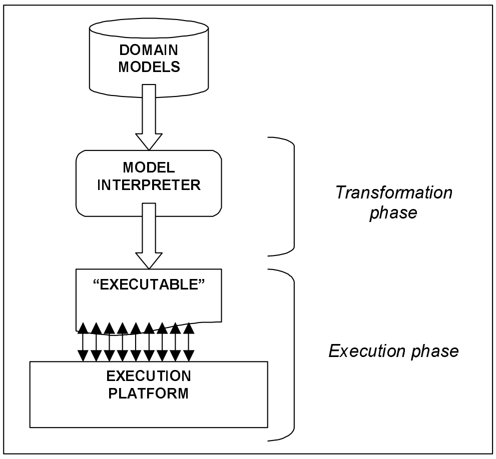

Figure 6: Assigning semantics to domain models

It seems natural that the semantics of domain models should also be captured in the metamodel of the domain. That is, $\boldsymbol{I_M}$ is to map a particular metamodel into a specific $\boldsymbol{I_D}$ 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 (i.e. $\boldsymbol{I_M}$) 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 [Karsai 98].

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 [CDIF 00] proposes the use of the four-layer modeling approach. The static semantics of UML [UML 97b] is specified using a similar approach, using UML and OCL as its own metalanguage. Metamodeling is an idea that has been addressed in many research workshops and programs (e.g. [Metamodeling 95] and [Asilomar 99]). Some of the relevant research activities and industry efforts are related to integrating data from various sources (e.g. MetaData coalition [MDC 00]).

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 field of CBS is already very large but growing, and it requires new tools and more formal techniques for the analysis and synthesis of these systems. Typically, a number of formal domain-specific modeling languages are needed for the precise specification and efficient design and implementation of computer-based systems. Our 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 generic modeling environment 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. Furthermore, if the metamodel captures the specification of the mapping of domain models into the ”instruction-set” of an execution platform, it can be used to automatically synthesize a transformation engine 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 [ISIS 00]. The tools typically feed various analysis and synthesis components that perform design-time analysis and/or synthesize executable systems. Our current work focuses on a more efficient method for mapping the abstract syntax of a metamodel onto the graphical idioms of the generic modeling environment and editing mechanisms within this environment. As indicated in the paper, the metamodel should contain specifications of model interpretation along with the syntax and ontology specifications. Active research is ongoing into the formation, representation, and interpretation of such specifications.

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

[Asilomar 99] 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

[CDIF 00] CDIF Meta Model documentation. http://www.metamodel.com/cdif-metamodel.html

[GME 00] Generic Modeling Environment documents, http://www.isis.vanderbilt.edu/projects/gme/Doc.html

[ISIS 00] Model-Integrated Computing project documents, http://www.isis.vanderbilt.edu/projects/

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

[Ledeczi 99] Ledeczi A., Maroti M., Karsai G., Nordstrom G.: "Metaprogrammable Toolkit for Model-Integrated Computing", Proceedings of the Engineering of Computer Based Systems (ECBS) Conference, Nashville, TN, March, 1999.

[MDC00] MetaData coalition. http://www.mdcinfo.com/

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

[Nordstrom 99a] Nordstrom G., Sztipanovits J., Karsai G., Ledeczi, A.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE ECBS’99, Nashville, TN, April, 1999.

[Nordstrom 99b] Nordstrom G.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Ph.D. Dissertation, Vanderbilt University, 1999.

[OCL 97] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997.

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

[Sztipanovits 97] Janos Sztipanovits and Gabor Karsai, ”Model-Integrated Computing,” *IEEE Computer*, pp. 110-112, April, 1997.

[UML 97a] UML Semantics, ver. 1.1, Rational Software Corporation, et al., September 1997.

[UML 97b] *UML Summary,* ver. 1.0.1, Rational Software Corporation, March, 1997