Metamodeling: An Emerging Representation Paradigm for System-Level Design

Alberto Sangiovanni-Vincentelli
University of California, Berkeley

Guang Yang
National Instruments

Sandeep Kumar Shukla
Virginia Polytechnic and State University

Deepak A. Mathaikutty
Intel

Janos Sztipanovits
Vanderbilt University

Editor's note:
The use of metamodeling in system design allows abstraction of concepts germane to a number of varying modeling domains, and provides the ability of exploiting meta-information for a variety of system design tasks such as analysis, verification, synthesis, and test generation. This article provides an overview of emerging metamodeling techniques and their applications.

—*Tim Cheng,* IEEE Design & Test *editor in chief*

Model-based design continues to gain wide acceptance in software, systems, and control engineering.[1] Clearly demonstrating this trend is the growing number of tools addressing diverse needs of different industries or engineering domains. In the automotive industry, Simulink and Stateflow are widely used in control system design and some aspects of code generation.[2] In the aerospace industry, Matrixx (http://www.ni.com/matrixx) is used for software architecture modeling and code generation. In the process and instrumentation industry, LabView (http://www.ni.com/labview) has a strong foothold. In SoC and multiprocessor SoC (MPSoC) design, SystemC (http://www.systemc.org) and related simulation and code generation tool suites are appearing. During the past five years, the Unified Modeling Language (UML) has been shifting from a design documentation notation to a model-based design platform for software engineering.[3] Today, modeling languages and model-based analysis methods and tools are being developed in all conceivable engineering domains.

Model-based methods' fundamental appeal is that they let system designers use abstractions matching their primary design concerns rather than be constrained by properties of the implementation technology they choose. Not surprisingly, a control engineer achieves more effective results by using a modeling language designed for defining and composing control systems (such as Simulink or Stateflow) than by using C++, which might be the controller's implementation language. In many engineering domains—and particularly in cyber-physical systems (http://www.cra.org/ccc/cps.php), where the systems are inherently heterogeneous—the model-based design process comprises a range of models representing different aspects of system behavior. The automated design process proceeds by refining, integrating, and analyzing these models in a complex flow. This complex, iterative model construction process is combined with model analysis to establish required properties and model transformations for integrating models, extracting information for analysis, or translating them into code. The richness of the model-based development process is formally captured by the platform-based design concept.[4,5]

The basic tenets of platform-based design are as follows:

- The design progresses in precisely defined abstraction layers.
- Each abstraction layer is defined by a design platform. A design platform represents a family of designs that satisfies a set of platform-specific constraints.
- Designs on each platform are represented by platform-specific design models. A complete design is obtained by a designer's creating platform instances via composing platform components and by mapping the platforms in the design flow onto subsequent abstraction layers.

Building, analyzing, and manipulating models are central to model-based design. Necessarily, the design process requires extensive tool support. Development of methods and tools are predicated on the precise specification of design platforms. Because a platform expresses a family of designs, the design platforms can be specified formally as modeling languages or as models of computation. A *model of computation* is a mathematical formalism that describes the computation and communication semantics of actors. *Actors* are basic computational elements that process data and communicate with one another to implement the intended computation. The computation semantics define how actors act, and the communication semantics define how they react. In a broad sense, MoCs address the behavioral modeling and compositional aspects in model-based design. Examples of common MoCs are finite-state machines, synchronous data flows, and discrete-event systems. Each has restricted behavioral semantics that make modeling—and, more important, model analysis and verification tasks—easier. Modeling languages define a representation method for expressing designs. Reflecting the richness of engineering design tasks, modeling languages span a wide range from informal graphical notations, such as the object modeling technique (OMT),[6] to formal textual languages, such as Alloy for software modeling.[7]

However, a single universal modeling language or MoC will not suit all domains. Thus, domain-specific modeling languages (DSML) and MoCs have emerged. Domain specificity creates two fundamental challenges: potential insularity and lack of communication across domains, and higher development costs. Metamodeling (http://www.metamodel.com/staticpages/index.php?page=20021010231056977) and metaprogrammable tools have emerged as responses to these challenges.[8-10] As research, methodology, and tool development for embedded-system design progress, a framework based on metamodels will unquestionably emerge as the standard. Accordingly, the goal of this article is to introduce metamodeling developments relevant to system-level design of electronic systems. We focus on approaches we have developed that form the basis for much of the research work in the US.

Metamodeling has two basic interpretations. The common interpretation refers to the modeling of modeling languages including the languages' concrete syntax (notations), abstract syntax, and semantics. Metamodels determine the set of valid models that can be defined with models' language and behavior in a particular domain. Generic functions in model-based design such as model building, model transformation, and model management are supported by metaprogrammable tools. The tools' core functions are independent from the particular DSMLs and can be instantiated using metamodels.

The second, less traditional interpretation relates to the use of models of computation for system design and has a strict semantics connotation. For this reason, we refer to this interpretation as a *semantics metamodel.* Although MoCs are powerful in capturing specific designs, embedded electronic systems are inherently heterogeneous. Hence, their modeling requires multiple MoC-specific models, thus making the overall system's analysis problematic because its behavior is not a priori expressible in a mathematical formalism that can be inferred from the components' MoCs. Metamodeling in this context is a way to uniformly abstract away MoC specificities while consolidating MoC commonalities in the semantics metamodel. This metamodeling results in a mechanism to analyze and design complex systems without renouncing the properties of the components' MoCs. This metamodeling concept lets us compare different models of computation, provide mathematical machinery to prove design properties, and support platform-based design. It forms the basis of several actor-based design environments such as Ptolemy II and Metropolis.[9,11]

Domain-specific modeling languages

Modeling languages play fundamental roles in model-based design. These roles can be divided into three categories:

- *Unified (or universal) modeling languages*, such as UML and Modelica, are broadly designed to offer adopters the advantage of remaining in a single-language framework, independent of application domain. Necessarily, the core language constructs are tailored more toward an underlying technology (for example, object modeling) rather than to a particular domain—even if extension mechanisms, such as UML profiling, allow customizability.
- *Interchange languages*, such as the Hybrid System Interchange Format (HSIF), permit model sharing across analysis tools (hybrid system analysis). Interchange languages are optimized for providing specific quantitative analysis capabilities in design flows by facilitating tool integration. The languages are optimized to cover concepts related to an analysis technology.
- *Domain-specific modeling languages* (DSMLs) specify a design platform, including the concepts, relationships, and well-formedness constraints linked to the application domain they address. They are optimized to be focused: the modeling language should offer the simplest possible formulation that is sufficient for the modeling tasks.

Noting these distinctions among modeling languages reveals an essential aspect of model-based design: we cannot assume that a single "universal" modeling language will fill all roles. A single language would defeat model-based design's purpose and fundamental advantage: creating abstraction layers in design flows. However, domain specificity creates two fundamental challenges:

- *Precise, formal specification of DSMLs.* Without rigorous specification of modeling languages, design flows disintegrate into loosely coupled regimes (such as software component architecture design, timing analysis, or system-level architecture design) formed around tool suites developed and used in relative isolation. Because practical design requires iteration across these regimes, the separation is costly, and it unavoidably leads to inconsistencies.
- *Tool infrastructure for model-based design*. Development of dedicated modeling and model analysis tool suites for rapidly changing application domains is cost prohibitive. Without resolving the dichotomy between domain specificity and reusability, model-based design will be restricted to slowly changing and relatively large application domains where a viable tool market can form.

Many relevant efforts have addressed these challenges, of which metamodeling is one major approach. An early metamodeling example is the case data interchange format (CDIF) standard developed for information interchange among CAD tools.[12] Progress in metamodeling has led to the emergence of metaprogrammable tools that have made DSML-based approaches practical.[8-10,13] To put it simply, metamodeling is the modeling of DSMLs. DSML modeling's purpose is to specify DSMLs in a formal, mathematically solid way using metamodeling languages. Generic functions in model-based design such as model building, model transformation, and model management are supported by metaprogrammable tools. These tools' core functions are independent from the DSMLs and can be instantiated using metamodels.

The current practice of specifying DSMLs covers a wide range of methods, from formal (which is beyond the scope of this article[14]) to informal. In the informal approach, specification is implicit: language constructs and notations are chosen to represent concepts familiar to users. Specifications take the form of explanations written in natural language (possibly interspersed with mathematical notations). At a minimum, writing down the semantic ideas and expressing their mathematical meaning reduces the chance for misunderstanding among developers. However, characterization of completeness or consistency of the specification is impossible as languages grow in size and complexity. This method's applicability is restricted to closed tool suites, in which the tool components are integrated by the tool vendors and where semantic consistency is maintained transparently to the user. In this approach, the model's behavior is frequently defined by a code generator, which translates the models to executable code. For example, Matrixx models are translated into Ada, which can be compiled further into executable code. At least three problems hinder the practicality of this approach:

- Understanding the semantics by observing the behavior generated by simulators or compiled code might be difficult, especially when deployed in highly integrated or distributed configurations.

- Changes made to the modeling language often require significant changes to the simulators or code generators, which often serve as the only fully detailed description of the DSML semantics. In addition, these languages and associated tools are frequently proprietary, which means that each new release of the tool suite might result in hidden changes in the modeling language's semantics.
- Source code templates are poor documentation, frequently tending toward incomprehensibility. Further, the unrestricted expressiveness of general-purpose programming languages often leads to undisciplined (but executable) specifications.

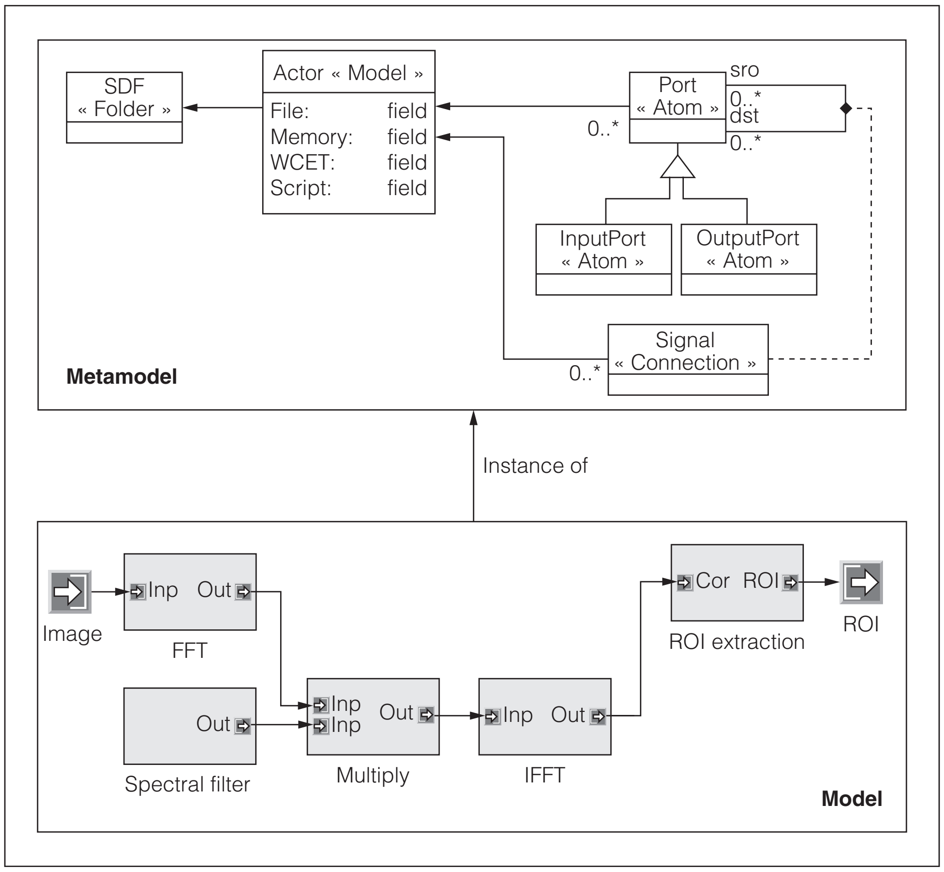


Figure 1. MetaGME metamodel and model instance for simple dataflow. (FFT: fast Fourier transform; IFFT: inverse FFT; ROI: region of interest.)

A common technique used extensively is to limit metamodeling to specify the abstract syntax. (The modeling language used for metamodeling is frequently UML class diagrams and Object Constraint Language or some other variations of the metamodeling languages;[15,16] for an overview, see Emerson et al.[17]) This approach represented a major advance and opened the possibility for developing metaprogrammable tool suites for model-based design, such as the Model-Integrated Computing (MIC; http://www.escherinstitute.org/Plone/tools/suites/mic) tool suite or the Eclipse modeling framework.[13]

For example, MetaGME (*GME* stands for generic metamodeling environment) is the metamodeling language for the generic modeling environment of the MIC tool suite.[18] MetaGME is a graphical language, and its concrete syntax derives from UML class diagrams augmented with UML class stereotypes. Figure 1 shows the metamodel and one of its possible instances for a simple dataflow language. According to the metamodel, a synchronous dataflow (SDF) can be modeled as a set of Actors with InputPorts and OutputPorts that are connected by Signals. The class stereotypes in the metamodel tell the GME modeling tool how to visualize the classes (for example, the Signal class is represented as a graphical connection between ports.) Only Ports can be connected by Signals. A well-formed instance of this metamodel shows the model of an image filter.

Although the metamodel and its relationship to the model is intuitively clear, this technique has two obvious problems:

- Well-formedness constraints captured in the metamodels limit the acceptable structure of the models in a domain. To precisely understand the meaning of metamodels, we must develop a mathematical model for the domains. This mathematical model provides *structural semantics* for the metamodeling languages.
- The model structures provide no help in understanding the models' behavior. We must develop a method to define the precise *behavioral semantics* for the modeling languages.

Full development of DSML structural and behavioral semantics is beyond the scope of this article. For more information, see Jackson and Sztipanovits.[19,20]

Metamodeling application: IP reuse

IP reuse requires that a library of IP blocks be available to the designer, along with a suitable CAD environment that lets the designer compose these IP components in a well-defined manner. Structural and behavioral constraints must be imposed on the way such composition can be carried out for the composed system to be correctly constructed. Metamodeling helps in designing such a constrained environment; meta-information on IP components helps in checking that the constraints are met; and metaprogrammable APIs allow for adding various model analysis and transformation tools to the environment for further processing, test generation, and verification.

Design reuse is the inclusion of previously designed components (design IP blocks) in the development or modeling of new software and hardware. *Verification reuse* is the ability to reuse existing verification models to verify a design at various abstraction levels (for example, microarchitecture level, RTL, and so on) by creating verification IP blocks at all these levels from a verification model at the highest abstraction level (such as the instruction set architecture [ISA] level).[21] Such reuse is best facilitated by model transformation techniques available with metamodeling. This kind of reuse allows coherence between the verification models at the various levels, and greater productivity by reducing the need to manually create verification IP for the design at each abstraction level.

IP reuse for SoC integration

For illustration, assume the availability of a library of C++-based (such as SystemC) IP blocks containing implementations of various standard blocks needed to create a system-level model (SLM) for a target SoC. The problem designers face is how to create an SLM of a new SoC by combining the IP blocks selected from the given library. Three possibilities while solving this problem may arise:

- the library does not have sufficient required and relevant IP blocks to build an SLM for the system under design,
- there are enough IPs that can be composed by generating the required programming glue to build the SLM, or
- multiple possible ways exist to combine the given IPs to create the model, which the designer must explore to build the most suitable model.

To decide which of these possibilities is the case—and in the event of the second or third possibility, to automate the composition process—the designer can use a *metamodeling-driven component composition framework*.[22] MCF lets designers

- create a visual module-, connection-, or bus-based template or a platform for the SoC;
- explore the SystemC IP library to automatically instantiate the template with real implementations based on type (structural and behavioral) matching; and
- automatically generate the programming glue logic for composing the selected IPs.

The essential issues and solutions that MCF was created to address are as follows:

- *System requirement specification:* MCF provides a metamodeling-based component composition language that lets designers specify the system requirements as a platform or as an abstract template.
- *IP metadata needed for reuse:* The IP metadata necessary to enable reuse includes compositional characteristics at both the structural and the behavioral levels, which are mined from the IP blocks in the IP library through automated extraction techniques.
- *IP metadata representation:* This meta-information is represented using XML schemas and populated in XML data structures to enable processing through XML parsers.
- *IP selection and composition:* MCF employs type-theoretic techniques and assertion-based verification to select IP implementations that structurally and behaviorally match the components in the visually defined platform. It automatically generates the necessary interfaces and transactors to compose the selected IP blocks and provide executable models as the end product.

MCF framework

MCF, built on top of MetaGME, captures the syntax and semantics of the visual composition language into a metamodel (see Figure 2).[22] The metamodel is developed using UML and OCL. The design environment lets a modeler visually construct platforms conforming to the metamodel's underlying restrictions. The MCF framework also includes libraries of compiled SystemC IP blocks, on which MCF performs reflection and introspection to identify, extract, and represent IP-specific metadata in XML structures.[21,23] It also has an *IP restrictor*, which captures constraints on IP blocks, letting the library have flexible or generic components. MCF provides a tool for automated selection and composition based on sound type-theoretic principles,[21] through its interactions with the IP library and the IP restrictor. Finally, an *interaction console window*—displayed when the software is invoked and which lets designers enter commands—ties in the different ingredients and techniques, enables design reuse and component composition in MCF. By instantiating the platform components and communication links or buses from the IP library and by synthesizing the necessary interfaces, we then develop an executable model for the SoC.[22]

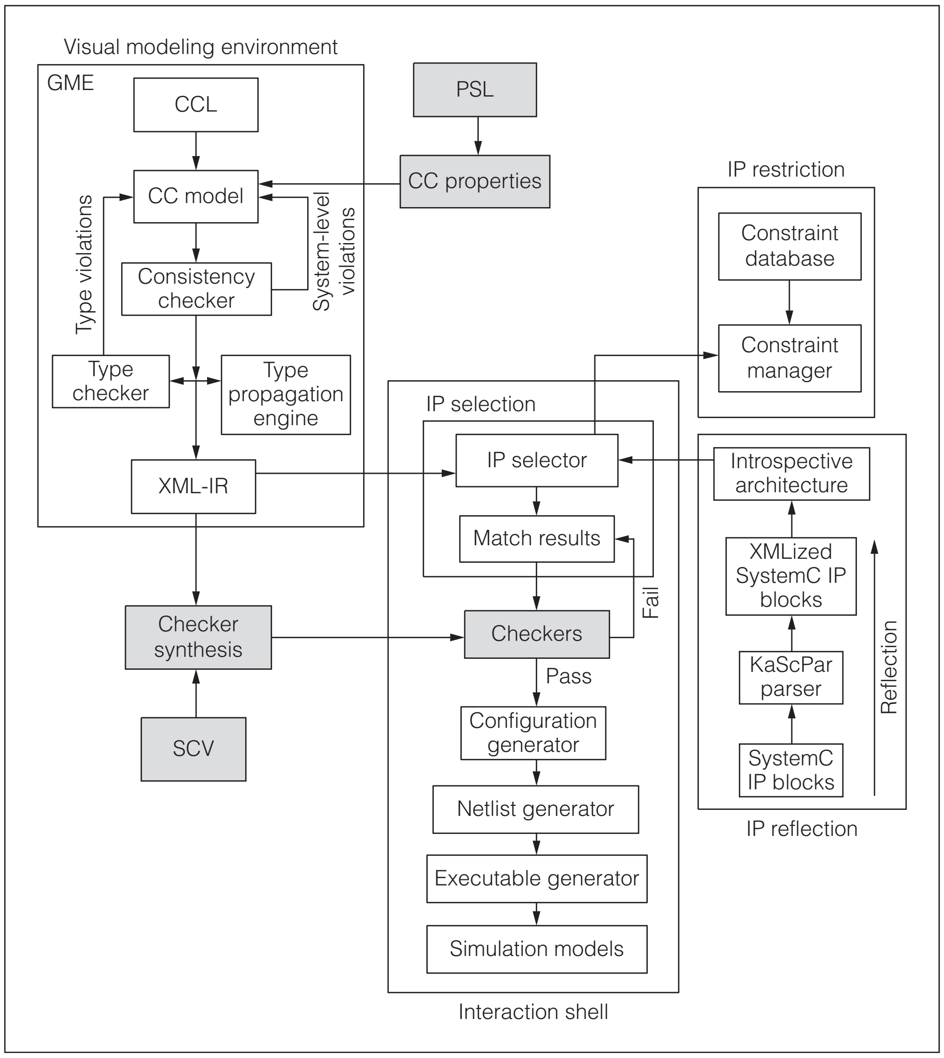


Figure 2. Metamodeling-driven component composition framework (MCF) design flow diagram.

Visual-modeling environment. The MCF visual-modeling environment built atop GME provides a visual language for describing component composition models (CCMs), which are platforms to be instantiated. The entities allowed in a CCM are components and media at varying abstraction levels such as RTL and transaction level (TL) with hierarchical descriptions. The visual language also enforces structural composition rules that describe the possible interconnections between the different entities that a modeler can instantiate and characterize to describe the system.[21,22] In addition, MCF allows some behavioral properties of the abstract components through the property specification language (PSL). Finally, the visual model and the behavioral properties are converted into an internal XML representation (XML-IR).

IP library. An MCF IP library is a collection of compiled SystemC IP cores. Given a library from which to perform IP selection, composition, and validation, MCF reflects composition-related metadata and creates an introspective architecture (that is, an architecture that lets the system query its own state and topology). The IP reflector extracts two kinds of metadata; one related to SystemC components and the other related to SystemC channels. The metadata on an RTL component contains the information (data types and bit widths) on I/O ports and clock ports. For a TL component or channel, the metadata contains information (function signatures that include function arguments and return types) on interface ports, clock ports, and interface access methods. An IP block's hierarchical structure is also extracted as a part of the metadata.

For details on the composition-related metadata extraction from SystemC, see Mathaikutty and Shukla.[21] The extracted information populates an XML data structure, which is part of the library. The library has appropriate APIs that allow introspection of the data structure. The main variants of the compositional metadata are as follows:

- RTL components,
- TL components and channels,
- TL interfaces,
- polymorphic components and channels,
- annotations such as pragmas or comments,
- type declaration and indirection from `typedef` and other similar constructs, and
- type restrictions necessary to constrain generic IP blocks.

A toolset can automatically extract this metadata from a SystemC IP block through metadata mining and uniform population of a data structure. The data structure can be queried through a set of APIs by MCF's other ingredients. In addition, user-specified behavioral metadata is provided as PSL properties capturing various temporal relationships between the signals at the component's I/O ports.

IP selector. The MCF's primary task is to select an appropriate IP block from the library that can be plugged into the platform to create possible executable models, allowing for rapid exploration of design alternatives. The IP selection problem in MCF and proposed techniques that perform automated abstraction and design visualization-based matching from an abstract architectural template of the system are available elsewhere.[21,22] The IP selector has various selection schemes:

1. For quick selections, the IP selector searches on the basis of nomenclature (version number, IP, and library name and distributor).
2. For RTL components, the selector searches on the basis of port-level structural type (data types).
3. For TL components and channels, the selector searches on the basis of interface-level structural type (function signatures and interface port specifics).
4. The selector can search on the basis of IP visualization: black-box, flattened, or hierarchical.
5. The selector can do a search using a mix of the first three schemes, in addition to opacity in the block-box visualization scheme, to obtain nine different selection schemes.
6. An exhaustive search can be made on the aforementioned nine selection schemes to obtain the ideal match.

The selection schemes exploit the composition-related metadata provided by the reflective-introspective capability of the IP library. Behavioral-type theory and type-matching algorithms as Talpin et al. have described could also be implemented,[24] but instead MCF does postcomposition behavioral conformity checking between composed components through automated test generation.[21]

Interaction console. The console defines the possible interactions of a modeler and a library engineer with MCF as well as outlines the task flow,[21] which can be divided into three stages: initialization, selection, and executable generation. During the initialization stage, the modeler creates the architectural template or the platform of the SoC, and the library engineer provides the necessary SystemC IP libraries. In the selection stage, the system automatically selects SystemC IP implementations that match virtual components in the template on the basis of sound structural type-theoretic techniques as well as assertion-based verification of behavioral properties.[22] As mentioned, the behavioral types are encoded as PSL properties of the components' interface signals. Automated test generation, followed by running a series of tests, checks for behavioral conformity. In the final, executable generation stage, the MCF software generates possible executable specifications of the target SoC by integrating the selected IP blocks and by implementing the necessary programming glue.

Note that although this component composition framework is created based on MetaGME and corresponding concepts of DSML-based metamodeling, we could take a different approach in creating a CCF with behavioral IP blocks using a semantics metamodeling approach. Although the relative pros and cons of the two approaches are beyond the scope of this article, such a comparative analysis will eventually be required as metamodeling finds wider use in creating tools for productivity gain through IP reuse.

Verification reuse: validation environment

A microprocess design flow starts, of necessity, with building models at a high abstraction level—namely, the ISA model. Subsequently, microarchitectural models, RTL models, and finally gate-level models are created through either manual transformations (from one level to the next) or automated synthesis techniques. Verification at each abstraction level requires modeling verification IP blocks at the appropriate levels. These verification IP blocks come in the form of simulators, test generators, coverage metrics, and test plans: collectively, they are *verification collaterals*. A possible reuse strategy is to use the models created for the various processor abstractions to derive different verification collaterals. A model reused to systematically derive the various verification tools and targets is a *generative verification IP*. The idea behind verification IP reuse is to model once and reuse many times. The alternative would be to derive the verification collaterals individually, which is tedious, error-prone, and labor-intensive.

Researchers extended the MCF metamodeling concept to create a *microprocessor validation environment* that provides a modeling framework and enables generative validation IP reuse and is related to Intel's microprocessor verification environment (see Figure 3).[25] Metamodeling-based verification (MMV) can model verification models at various abstraction levels (system, architectural, and microarchitectural levels). MMV provides a unified language to model verification IP blocks at all abstraction levels, and verification collaterals such as testbenches, simulators, and coverage monitors that can be generated from these models, thereby enhancing verification reuse. The metamodeling-driven modeling framework first provides the syntax and semantics needed to uniformly capture a processor's various modeling abstractions. Metamodeling enforces the rules that restrict the models at various abstraction levels. Metamodeling also provides a way to express rules that enforce well-defined model construction in each abstraction level and consistency across abstractions. A visual editor on top of MetaGME facilitates processor modeling and enforces the modeling rules through checkers during design. MMV's generative validation IP reuse capability allows processor models to be translated into executables and other verification collaterals.[21,25]

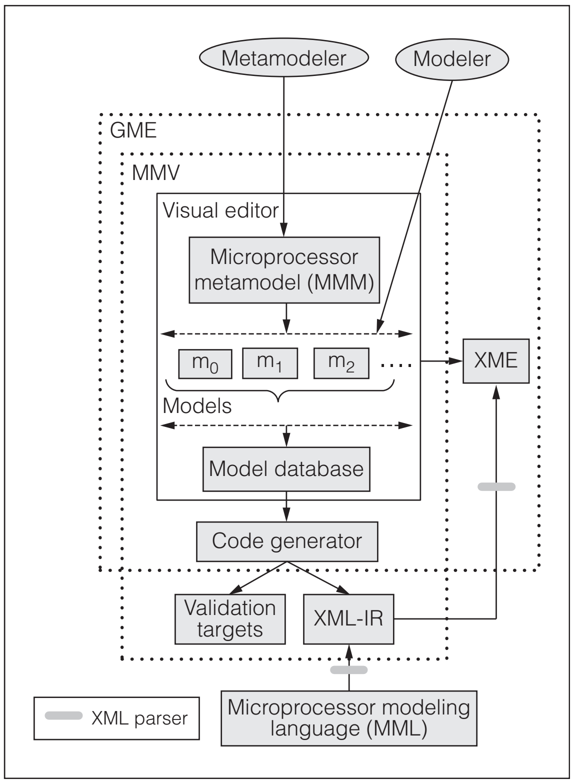


Figure 3. Microprocessor validation environment.

Semantics metamodels

Semantics metamodels, which are generalizations of models of computation, provide an alternative view of metamodeling and its use in system design.[26,27]

MoCs and semantics metamodels

To use formal models for ensuring safe and correct designs, the designer must understand the interaction between diverse formal models. There is a broad range of potential design formalizations, but most tools and designers describe a design's behavior as a relation between a set of inputs and a set of outputs. This relation may be informal, even expressed in natural language. The notion of MoCs was introduced to denote at one time the mathematical properties of a representation and its operational aspects (see Savage, for example[28]). Unfortunately, there does not seem to be a unique, precise mathematical definition of the concept, but the intuitive notion presented here has been rather uniformly used in the literature.

Recall that a language is a set of symbols, rules for combining them (its syntax), and rules for interpreting combinations of symbols (its semantics). Two approaches to semantics have evolved: operational and denotational. *Operational semantics*, which date back to Turing machines, give the meaning of a language in terms of actions taken by some abstract machine, and are typically closer to the implementation. *Denotational semantics,* first developed by Scott and

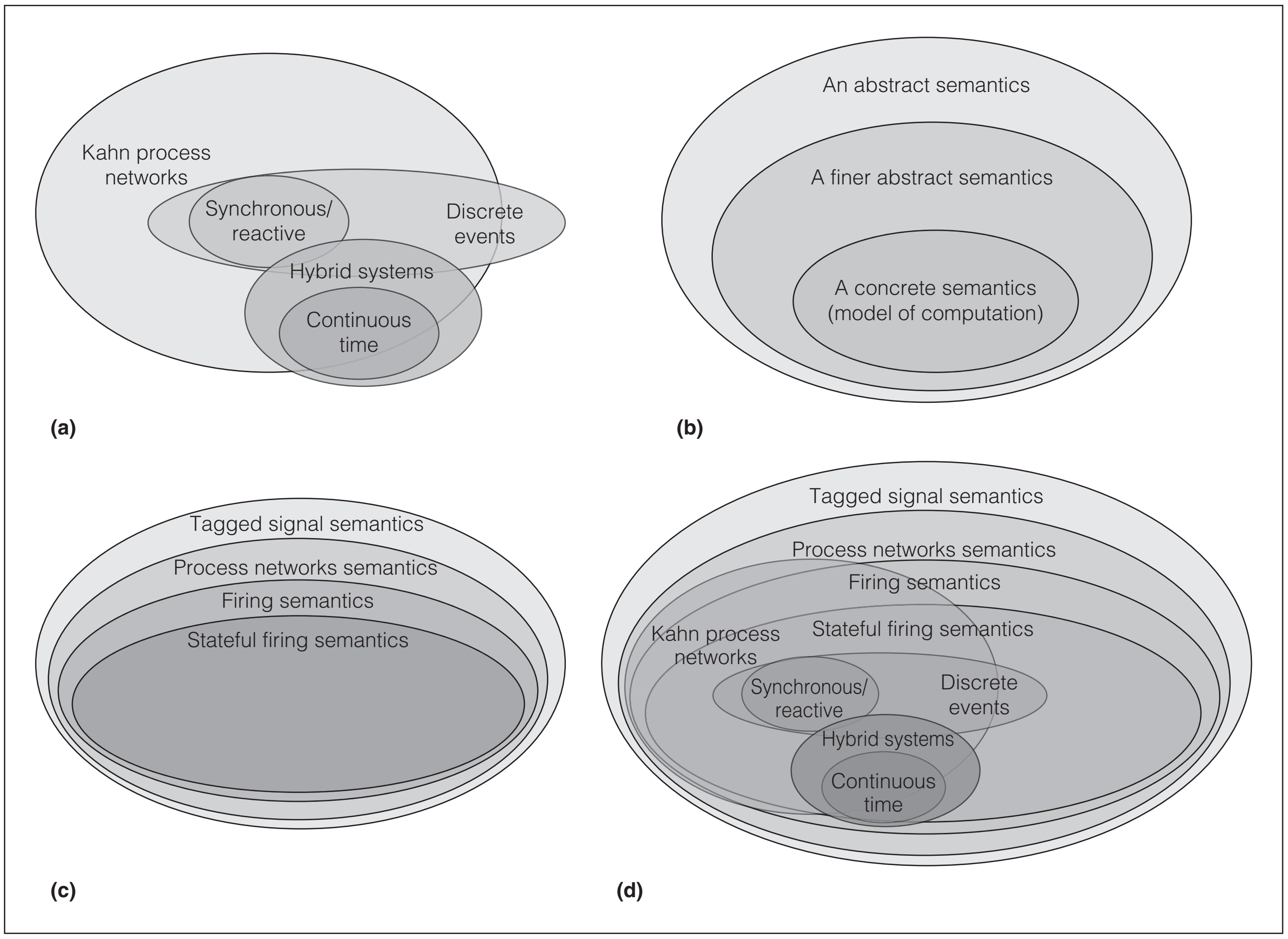


Figure 4. Abstract semantics and model of computation (MoC): containment relationships among MoCs (a), abstract and concrete semantics (b), some abstract semantics and their relationship (c), and abstract semantics and its relationship with standard MoCs (d). (These diagrams are renditions of slides presented by Edward Lee.[30])

Strachey,[29] give the meaning of the language in terms of relations. How the abstract machine in an operational semantics can behave is a feature of what we call the MoC underlying the language. The kinds of relations possible in denotational semantics is also a feature of the MoC. Other features include communication style, how individual behavior is aggregated to make more complex compositions, and how hierarchy abstracts such compositions.

Many MoCs have been defined, resulting from the immaturity of the field and also from fundamental differences: the best model to use is a function of the design. Examples of MoCs include finite-state machines (FSMs), static dataflow (SDF), Kahn process networks (KPNs), synchronous reactive (SR), discrete event (DE), and continuous time (CT).[26] SDF is a natural MoC to capture streaming and multimedia applications, CT is useful for capturing physical phenomena, and many safety-critical applications are expressed with SR semantics.

A natural ordering of MoCs is inferred by the relation of *behavior containment.* Figure 4a shows the containment relationships of some of the most well-known MoCs. For example, a KPN contains all the behaviors of an SDF model, hence, it is more general. The SDF model is more restrictive, but more powerful properties such as schedulability can be inferred through theoretical analysis, while in the case of KPN, many properties are difficult to verify and might require extensive computation.

For example, consider the property of *determinate behavior*—that is, the fact that a system's output depends only on its inputs and not on some internal, hidden choice. Any design described by a dataflow network is determinate, and so this property need not be checked. If the design is represented by a network of FSMs, we can assess determinacy by inspecting the state transition function. In both cases, the use of a particular MoC to capture a design helps in assessing properties without running expensive tests. There is indeed a trade-off between expressivity (generality) and the formal properties that can be associated with a model. In general, the more restricted the model, the more can be said about its mathematical properties.

In many situations, using a unique general model of computation for the entire design is equivalent to giving up any possibility of property checking without resorting to extensive simulation. On the other hand, if we use the most restrictive model in terms of behavior for every part of the design, we can leverage the model's richness but we must determine a different way to assess the design's overall properties. Indeed, the heterogeneous nature of most embedded systems makes multiple MoCs a necessity. In addition, during the design process, the abstraction level, detail, and specificity in different parts of the design vary. The skill sets and design styles that different engineers use on the project are likely to differ. The net result is that, during the design process, many different specification and modeling techniques will be used.

The challenge is how to combine heterogeneous MoCs and determine what the composition's behavior is. Unfortunately, the semantics of each MoC are incompatible. Hence, it is not even clear what composition means unless this meaning is explicitly specified. A way to solve this problem is to embed the detailed models into a framework that can understand the models being composed. A theoretical approach to this view, which is well beyond the scope of this article, can be found in the work of Burch et al.,[31] who used an abstract algebra approach to define the interactions among incompatible models.

In some sense, we are looking at an abstraction of the MoC concept that can be refined into any of the MoCs of interest. We call this abstraction *abstract semantics,* first introduced by Lee et al.[32] In Figure 4b, a Venn diagram expresses the abstraction relation between a class of abstract semantics and a particular MoC characterized by a concrete semantics. The inspiration on how to define the abstract semantics comes from the consideration that MoCs are built by combining three largely orthogonal aspects: sequential behavior, concurrency, and communication.

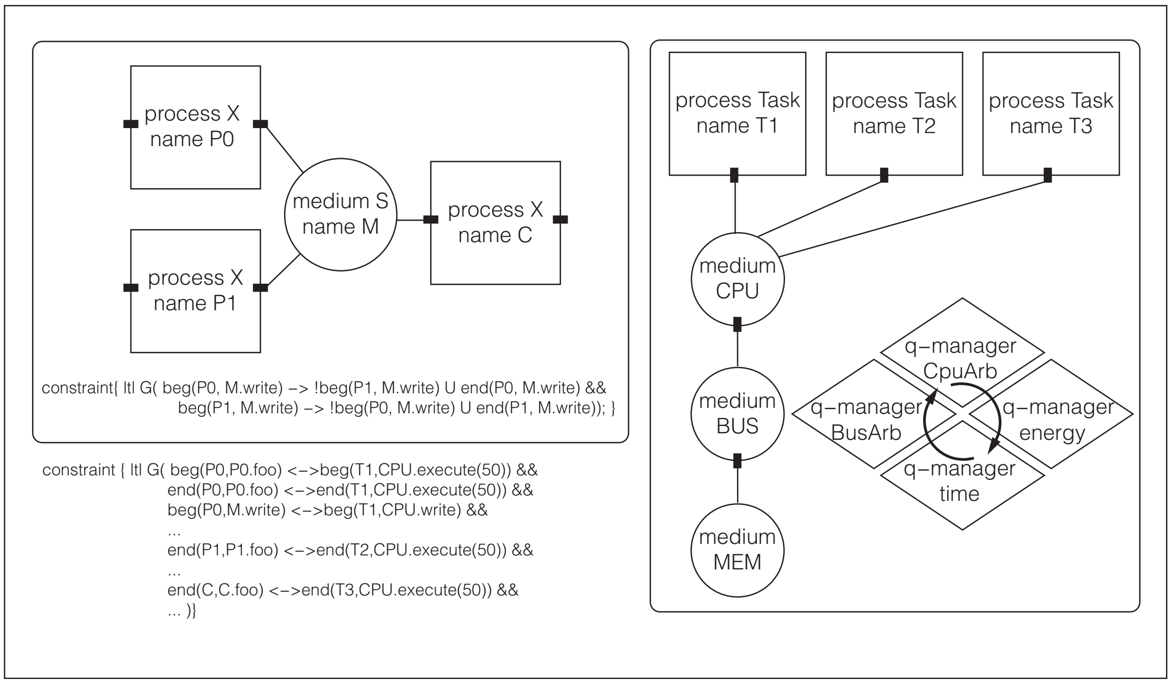
Similar to the way that a MoC abstracts a class of behavior, abstract semantics abstract the semantics of various MoCs. There are many ways of abstracting MoCs, as Figure 4c indicates. Each set of abstract semantics represents a model that is then called a *semantics metamodel.*

In previous work, a very general semantics metamodel has been defined: the *tagged signal model* (TSM), also called the Lee–Sangiovanni-Vincentelli (LSV) model.[27] Its semantics is denotational, because it was introduced to compare the MoCs that have been in use and to possibly derive new ones.

In TSM, the basic entity is an *event* $\in T \times V$, where T is a set of values and V is a set of tags. Tags could be used to establish ordering relations, such as time. A *signal s* is a set of events, whereas a *functional signal* is a function from T to V. A *process* with *n signals* is a set of possible *behaviors*, where each *behavior* $\in S^n$, and S is the set of all signals. When composing *processes*, their shared *signals* are intersected to derive the overall process. The model is extremely simple yet powerful enough to express concrete MoCs. Intuitively, a TSM consists of *processes* that run concurrently; the constraints imposed on their shared signals' tags define communication among them. Tags can represent a broad range of annotated relations, such as total orders in timed systems, partial orders in untimed systems or nonorder bearing cost information.

We used the concepts of abstract semantics and related abstract metamodels to design two general frameworks for system-level design: the Metropolis design environment and Ptolemy II.[8,9] Because the TSM is a denotational metamodel, we needed to derive operational versions of it based on the TSM but with somewhat finer abstract semantics. Called *process networks metamodel,* the TSM "contains" the semantics, as Figure 4b shows, in the sense that all models conforming to the process networks' abstract semantics also conform to the TSM's abstract semantics.

Other design environment modeling approaches can be referred to still finer abstract metamodels, as Figure 4b indicates. The name we gave to these abstract metamodels reflects the basic mechanism implemented: in the *firing abstract semantics*, the dataflow firing mechanism is abstracted and inserted into the more general abstract process network metamodel. The *stateful firing abstract semantics* capture the notion of state and can be used to abstract all state-based models of computation. Figure 4d shows the containment relationship of the abstract semantics with respect to standard MoCs.



```
constraint{ ltl G( beg(P0, M.write) −> !beg(P1, M.write) U end(P0, M.write) &&
                   beg(P1, M.write) −> !beg(P0, M.write) U end(P1, M.write)); }
```

```
constraint { ltl G( beg(P0,P0.foo) <−>beg(T1,CPU.execute(50)) &&
                    end(P0,P0.foo) <−>end(T1,CPU.execute(50)) &&
                    beg(P0,M.write) <−>beg(T1,CPU.write) &&
                    ...
                    end(P1,P1.foo) <−>end(T2,CPU.execute(50)) &&
                    ...
                    end(C,C.foo) <−>end(T3,CPU.execute(50)) &&
                    ... )}
```

Figure 5. Functional model, architectural model, and their mapping.

Metropolis and its metamodel

The *Metropolis metamodel* (MMM) implements the process network abstract semantics and describes all the ingredients advocated by the platform-based design (PBD) methodology, namely functionality and architecture across MoCs and abstraction levels, and the mapping relations between them.[11,33] The precise MMM semantics enables Metropolis to specify complex systems without ambiguity, and to perform synthesis and formal analysis in addition to simulation on the systems. Metropolis and its metamodel have been used extensively in system design, including automotive systems, ICs, and buildings—with particular emphasis on multiple levels of abstraction and mapping of functionality to architecture. Examples of its use can be found elsewhere.[5,34-37]

As shown in Figure 5, the MMM describes a functional model or an architectural model as a set of objects that concurrently take actions while communicating with one another. Each such object is termed a *process* and is associated with a sequential program called a *thread*. A process communicates through ports, where a port is specified with an interface, declaring a set of methods that can be used by the process through the port. In general, we can have a set of implementations of the same interface, and objects that implement port interfaces are *media*. Any medium can be connected to a port if it implements the port interface. Media do not have their own threads. Computation and communication are usually modeled separately by *processes* and media. On top of them, *quantity managers* are introduced for two purposes: performance modeling and scheduling. They can respond to quantity annotation requests sent by processes or media. After iterative resolution, quantity managers will associate the resulting annotations (tags) with the requesting behavior to reflect the model performance or scheduling. Finally, the MMM supports declarative constraints orthogonal to the imperative models. The constraints could be used either as properties that should be satisfied by the imperative model, which can be verified by validation tools, or as a refinement of the imperative behavior, in which case constraints and models both shape the desired behavior.

The MMM is generic and expressive enough to model different MoCs so that they can all be described and manipulated in a unique framework. To make modeling easier, it is usually wise to provide well-designed and well-tested MoC platforms, such that designers can simply extend the components in the platform to create their own model with that MoC semantics. Of course, if needed, users can always add MoCs, expressed in the MMM.

A MoC platform frequently has three major components: computation, communication, and coordination. Sometimes, in simple MoCs, the coordination component is absorbed by the computation and communication components. A few interesting MoCs have been implemented and tested as part of the Metropolis MoC platform library, such as SDF, synchronous systems, and fault-tolerant dataflow (FTDF). When using a particular MoC, designers must import the platform for that MoC, and inherit the computation; the communication component; and the coordination component, if any.

Computation. A model in a synchronous MoC contains computing entities connected via point-to-point single-storage channels. Based on its functionality, a computing entity can be classified as a Moore entity or a Mealy entity: the former generates outputs solely on the basis of its internal states, whereas the latter also depends on its current inputs. For Moore entities, their outputs are not available to others until the next computation cycle. Because the computing entities are synchronous, they all run once per cycle. However, to make the semantics a little simpler, in this MoC, we do not enforce any scheduling among entities even though data dependencies might exist. As a consequence, the model is nondeterministic, and the behavior can vary because of different execution orders of the entities.

In a MoC platform, a computation component is usually modeled by a process. With a synchronous MoC, the computation is easy—reading data through its input ports, performing computation, and writing data to its output ports. The synchronous MoC platform captures these three tasks in a process; the read-data, compute, and write-data become three methods in the process; and the thread of the process keeps calling these methods in that order. When designers model a synchronous computation component, they will only need to inherit the process defined in the platform, configure the number of input and output ports, and provide the component computation by overriding the computational method defined in the platform. This style of process skeleton definition and inheritance are the prototypical way of defining and instantiating MoCs in Metropolis.

Communication. Communication plays a crucial role in MoCs. It defines how components interact with one another, the distinguishing aspect of different MoCs. For instance, in KPNs, communication uses unbounded FIFO channels with blocking read and nonblocking write semantics, which guarantees a deterministic execution. In the rendezvous mechanism, communication blocks the sender or the receiver until both are ready to receive and send. In a synchronous MoC, communication takes the semantics of one-place FIFO processing with nonblocking write and nonblocking read.

In the Metropolis MoC platforms, communication is modeled with media. For the synchronous MoC, a medium consists of a storage variable and a flag. It also provides a read-and-write function. When the write function is called, it saves the data into storage and validates the flag. When the read function is called, it checks the flag first. If the flag is valid, it returns the stored data, then invalidates the flag; if the flag is invalid, it returns special data. Unlike computation components, in the synchronous MoC, the communication model needs no further modification. It can be simply instantiated and set up to connect computation components.

Execution order. In Metropolis MoC platforms, a model's execution order can be modeled by quantity managers. Processes or media can insert quantity annotation requests anywhere in their behavior description. Whenever quantity requests are executed, execution sends the requests to quantity managers, stops, and waits for quantity annotation results. Depending on the annotation results, these quantity managers can either suspend or resume execution. This mechanism effectively achieves execution-order scheduling, and significantly increases the reusability of the scheduling algorithms by capturing them with dedicated entities: quantity managers.

In the synchronous MoC, each computation component consists of three operations: read inputs, compute, and write outputs. To satisfy the synchronous semantics, all components must execute the three operations once per cycle and in that order. In addition, if there are any Moore components, their write-outputs operations must be held to the end of the current cycle after all other components finish their operations, because the outputs of Moore components are only visible in the next cycle.

To introduce the cyclic behavior, all processes must be synchronized after the three operations. Therefore, it is natural to add a barrier after the write-outputs operations. If there are any Moore components in the system, their write-outputs operations must be delayed until the end of the cycle. Therefore, for Moore components, a second barrier is needed before the write-outputs operations. The two barriers are modeled by a quantity annotation request. When sending the requests, additional information such as the identity of the barrier can be passed along. Based on the requests, the quantity manager can decide which requesting processes should proceed and which should wait. When creating a synchronous MoC model, such a quantity manager should be instantiated and connected to all requesting processes, with no further modification needed.

Ptolemy II

The goal of the Ptolemy project (http://ptolemy.berkeley.edu) is to provide a framework for heterogeneous modeling and execution of complex functional systems. Ptolemy II is based on an actor-oriented paradigm that can be interpreted as process-network abstract semantics.[9]

Actors are concurrent components that communicate through interfaces called ports. Actors can be atomic or composite (containing other actors). Relations define the interconnection between these ports, thereby also defining the communication structures between actors. When ports are connected, *channels* are established. Unlike Metropolis, where communication is performed by calling media-provided interface methods, Ptolemy II is not allowed to incur such transfers of control flow: only data is passed through ports. Each actor can run in its own thread, or all the actors can run sequentially in a single thread. Therefore, another entity—a director—must be provided to handle actors' scheduling and communication.

Actors and directors can be used in Ptolemy II to build a rich set of domains—that is, refinements of the metamodel into a more specific metamodel. The framework comes with predefined MoCs, including SDF, SR, DE, CT, and PN. Actors can be defined in a "generic" form to be used with a variety of directors, which simplifies domain construction. These actors are *domain polymorphic;* examples include data sources and sinks, arithmetic operators, logic operators, and signal-processing operators.

Computation. Actors contain three executable phases: setup, iterate, and wrap-up. Each phase can have finer-grained subphases.

- *Setup.* The setup phase has pre-initialize and initialize subphases. Pre-initialization usually handles structural information, such as instantiating dynamically created actors, deciding port widths, and creating receivers associated with input ports. Initialization sets up parameters, resets local states, and generates initial tokens. Usually, pre-initialization is performed exactly once for an actor at the beginning of its life cycle. Initialization is performed once after pre-initialization, but can be run again if the semantics require reinitialization.
- *Iterate.* In the iterate phase, actors perform atomic executions. An iteration is a finite computation that leads the actor to a quiescent state. The MoC semantics determine how the iteration of one actor relates to that of another. To coordinate the iterations among actors, an iteration is further broken down into prefire, fire, and postfire. Prefire checks the preconditions for the actor to execute, such as the presence of sufficient inputs to complete the iteration. The fire subphase usually does the actor's computation, which might involve reading inputs, processing data, and writing outputs. Postfire updates the actor's persistent state. The separate fire and postfire subphases ensure that the current computation result would not propagate to other actors, which is essential to support fixed-point iteration in some MoCs, such as SR and CT. These MoCs compute the fixed point of actor outputs while keeping the actor states unchanged. To reach the fixed point, multiple firing of each actor can be performed before the states are updated by postfire.
- *Wrap-up.* At the end of the execution, wrap-up runs once to clean up, for example, resources that were allocated to actors during execution.

Communication. Actors communicate with one another by sending and receiving data through ports. The communication mechanism is implemented using receivers contained in input ports. Because of different communication semantics, receivers might implement FIFO queues, mailboxes, proxies for a global queue, or rendezvous points. Receivers are created in pre-initialization for each actor. The domain polymorphism of actors is partially realized through the dynamic creation of receivers for different domains.

Execution order. A MoC defines the communication semantics and the execution order among actors.

This is realized through implicit (invisible to designers) receivers and directors. A director controls the execution order of actors in the same composite actor. When a composite actor is fired, the director inside the composite actor fires the actors of the contained model. For instance, in the SDF domain, the director statically computes scheduling. During execution, the director follows the scheduling and invokes each actor one by one. In the DE domain, a time line is maintained. Whenever an actor generates an event, the event is sorted in the time line according to its time stamp. The director always removes the event with the lowest time stamp and fires the event's destination actor.

Metamodeling without directors. Benveniste et al. provided an interesting extension and formalization for Ptolemy-style metamodeling in which the concepts of actors and directors were cast in a TSM framework but with a version of KPNs, which constitutes finer abstract semantics than the process-network abstract semantics.[38] In particular, although the MoC semantics are naturally expressed in terms of actors and directors, directors are not essential for semantics purposes and are justified only by simulation efficiency purposes. Benveniste et al. then claimed that, at least semantically, directors do not play an important part in defining a MoC.[38]

Related approaches

Other efforts that can be related to the concept of semantics metamodels are EWD, SystemCH, and HetSC. EWD is a customizable multi-MoC modeling environment based on the GME methodology.[39] EWD defines three phases in its flow: model construction, parsing, and code generation. For a MoC domain, a visual modeling language can be designed using GME. Applications can then be created in the model construction phase. The parsing phase produces an XML representation for the application, and code generation can translate it into models in languages such as SML, Haskell, or SMV for various analysis purposes.

SystemCH and HetSC are based on SystemC—a set of libraries written in C++. SystemC provides modeling capabilities for time, concurrency, and synchronization. As a system design language, it has gained momentum in recent years. However, it is limited with its discrete event semantics. SystemCH[40] and HetSC (http://www.teisa.unican.es/HetSC) have extended SystemC by adding the support of other MoCs, but their approaches are completely different.

SystemCH directly modifies the SystemC kernel (a part of the library) by adding more modeling constructs for different MoCs. At the same time, it also integrates MoC-specific tools into the kernel, such as the scheduling tool for SDF. On the other hand, HetSC does not change the library but extends it for other MoCs. In addition, HetSC also provides converter channels among different MoCs. Therefore, from a modeling perspective, HetSC is closer to Metropolis than SystemCH. However, because the Metropolis metamodel defines more rigorous semantics than SystemC, the analysis and realization of MMM is much easier.

METAMODELING IS BECOMING an important foundation of next-generation design methodology and tools for system-level design. For this reason, it is important to point to relevant past and present research. We have no doubts that metamodeling will find its way into industrial-strength design tools as system-level design becomes increasingly complex, encompassing multiple-semantics domains. In this respect, metamodeling will be a key concept in the development of design methods for next-generation embedded systems in which the physical is codesigned with digitally controlled, cyber-physical systems. ■

Acknowledgments

We gratefully acknowledge the many influential discussions with colleagues and students—in particular, Edward Lee, who inspired the discussion about abstract semantics and metamodels; and Albert Benveniste, Ethan Jackson, Roberto Passerone, Yosinori Watanabe, Luciano Lavagno, and Felice Balarin. This research has been supported by the Marco Focus Center Research Program (FCRP) Gigascale Systems Research Center (GSRC), the ArtistDesign network of excellence, a Semiconductor Research Corp. Integrated Systems Task, Air Force Office of Scientific Research (AFOSR), and the Component-Based Embedded Systems Design Techniques (Combest) STREP (a European funding program).

■ References

1. G. Nicolescu, *Model-Based Design for Embedded Systems,* CRC Press, 2009 (to appear).
2. J.B. Dabney and T.L. Harman, *Mastering Simulink,* Prentice Hall, 2004.

3. G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide,* 2nd ed., Addison-Wesley, 2005.
4. K. Keutzer et al., ''System Level Design: Orthogonolization of Concerns and Platform-Based Design,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 19, no. 12, 2000, pp. 1523-1543.
5. A. Sangiovanni-Vincentelli, ''Quo Vadis, SLD? Reasoning about the Trends and Challenges of System Level Design,'' *Proc. IEEE,* vol. 95, no. 3, 2007, pp. 467-506.
6. W. Premerlani et al., *Object-Oriented Modeling and Design,* Prentice Hall, 2005.
7. D. Jackson, ''Alloy: A Lightweight Object Modelling Notation,'' *ACM Trans. Software Engineering and Methodology,* vol. 11, no. 2, 2002, pp. 256-290.
8. F. Balarin et al., ''Metropolis: A Design Environment for Heterogeneous Systems,'' *Multiprocessor Systems-on-Chips,* W. Wolf and A. Jerraya, eds., Morgan Kaufmann, 2004.
9. C. Brooks et al., eds., *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II),* tech. report UCB/ERL M05/21, Univ. of California, Berkeley, 2005.
10. M. Emerson, S. Neema, and S. Sztipanovits, ''Metamodeling Languages and Metaprogrammable Tools,'' 2007; http://www.isis.vanderbilt.edu/node/3978.
11. F. Balarin et al., ''Metropolis: An Integrated Electronic System Design Environment,'' *Computer,* vol. 36, no. 4, 2003, pp. 45-52.
12. M. Imber, ''The CASE Data Interchange Format (CDIF) Standards,'' *Software Engineering Environments,* F. Long ed., Ellis Horwood Series in Information Technology, Ellis Horwood, 1991, pp. 457-474.
13. M. Emerson, S. Neema, and J. Sztipanovits, ''Metamodeling Languages and Metaprogrammable Tools,'' *Handbook of Real-Time and Embedded Systems,* I. Lee, J. Leung, and S.H. Son, eds., CRC Press, 2006.
14. See D. Harel and B. Rumpe, ''Modeling Languages: Syntax, Semantics and All That Stuff,'' tech. report MCS00-16, Weizmann Inst. Science, 2000. This article was later popularized by the authors in D. Harel and B. Rumpe, ''Meaningful Modeling: What's the Semantics of 'Semantics'?'' *Computer,* vol. 37, no. 10, 2004, pp. 64-72.
15. *Unified Modeling Language: Superstructure* v2.0, 3rd revised submission to OMG RFP, tech. report, Object Management Group, 2003.
16. *Object Constraint Language* v2.0, tech. report, Object Management Group, 2006.
17. M. Emerson, J. Sztipanovits, and T. Bapty, ''A MOF-Based Metamodeling Environment,'' *J. Universal Computer Science,* vol. 10, no. 10, 2004, pp. 1357-1382.
18. G. Karsai et al., ''The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design,'' *Proc. IEEE Joint Conf. CCA, ISIC, and CACSD,* IEEE Press, 2006, pp. 50-55.
19. E.K. Jackson and J. Sztipanovits, ''Towards a Formal Foundation for Domain Specific Modeling Languages,'' *Proc. 6th ACM Int'l Conf. Embedded Software* (EMSOFT 06), ACM Press, 2006, pp. 53-62.
20. E. Jackson and J. Sztipanovits, ''Formalizing the Structural Semantics of Domain-Specific Modeling Languages,'' *J. Software and Systems Modeling,* 2009 (to appear).
21. D.A. Mathaikutty and S.K. Shukla, *Metamodeling Driven IP Reuse for System-on-a-Chip Integration and Verification,* Artech House, 2009.
22. D.A. Mathaikutty and S.K. Shukla, "MCF: A Metamodeling-Based Visual Component Composition Framework," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 7, 2008, pp. 792-805.
23. D.A. Mathaikutty and S.K. Shukla, ''Mining Metadata for Composability of IPs from SystemC IP Library,'' *Design Automation for Embedded Systems,* vol. 12, no. 1, 2008, pp. 63-94.
24. J.P. Talpin et al., ''A Behavioral Type Inference System for Compositional System-on-Chip Design,'' *Proc. Applications of Concurrency in System Design* (ACSD 04), IEEE CS Press, 2004, pp. 47-56.
25. D.A. Mathaikutty et al., ''MMV: A Metamodeling-Based Microprocessor Validation Environment,'' *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 4, 2008, pp. 339-352.
26. S. Edwards et al., ''Design of Embedded Systems: Formal Models, Validation, and Synthesis,'' *Proc. IEEE,* vol. 85, no. 3, 1997, pp. 366-390.
27. E. Lee and A. Sangiovanni-Vincentelli, ''A Framework for Comparing Models of Computation,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 17, no. 12, 1998, pp. 1217-1229.
28. J.E. Savage, *Models of Computation: Exploring the Power of Computing,* Addison-Wesley, 1998.
29. J.E. Stoy, *In Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.
30. E.A. Lee, ''Concurrent Models of Computation for Embedded Software,'' tech. memo UCB ERL M05/2, Univ. of California, Berkeley, 4 Jan. 2005; http://ptolemy.eecs.berkeley.edu/papers/05/Lee_Lectures/, pp. 317-325.
31. J.R. Burch, R. Passerone, and A.L. Sangiovanni-Vincentelli, ''Refinement Preserving Approximations for the Design and Verification of Heterogeneous Systems,''

Formal Methods in System Design, vol. 31, no. 1, Aug. 2007, pp. 1-33.

32. E.A. Lee et al., ''Actor-Oriented Design of Embedded Hardware and Software Systems,'' *J. Circuits, Systems, and Computers,* vol. 12, no. 3, 2003, pp. 231-260.
33. Metropolis Design Team, ''The Metropolis Meta Model v0.4,'' tech. memo UCB/ERL M04/38, Univ. of California, Berkeley, 14 Sept. 2004.
34. A. Bonivento, L.P. Carloni, and A.L. Sangiovanni-Vincentelli, ''Platform-Based Design of Wireless Sensor Networks for Industrial Applications,'' *Proc. Design, Automation and Test in Europe Conf.* (DATE 06), IEEE CS Press, 2006, pp. 1103-1107.
35. S. Kanajan et al., ''Exploring Trade-off's between Centralized versus Decentralized Automotive Architectures Using A Virtual Integration Environment,'' *Proc. Design, Automation and Test in Europe Conf.* (DATE 06), IEEE CS Press, 2006, pp. 548-553.
36. D. Densmore, A. Donlin, and A. Sangiovanni-Vincentelli, ''FPGA Architecture Characterization for System Level Performance Analysis,'' *Proc. Design, Automation and Test in Europe Conf.* (DATE 06), IEEE CS Press, 2006, pp. 1-6.
37. A. Pinto et al., ''Synthesis of Embedded Networks for Building Automation and Control,'' *Proc. Am. Control Conf.* (ACC 08), IEEE Press, 2008, pp. 920-925.
38. A. Benveniste et al., ''Actors without Directors: A Kahnian View of Heterogeneous Systems,'' *Proc. Hybrid Systems: Computation and Control* (HSCC 09), LNCS 5469, Springer, 2009, pp. 46-60.
39. D.A. Mathaikutty et al., ''EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Environment,'' *Proc. ACM Trans. Design Automation of Electronic Systems,* vol. 12, Dec. 2007, article 33.
40. H.D. Patel and S.K. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling: A Framework for Multi-MoC Modeling,* Springer, 2004.

Alberto Sangiovanni-Vincentelli holds the Buttner Chair of Electrical Engineering and Computer Science at the University of California, Berkeley. His research interests include system-level design, embedded and hybrid systems, and EDA. He has a Dr Eng in electrical engineering and computer sciences from Politecnico di Milano. He is a Fellow of the IEEE, and is a member of the National Academy of Engineering and the ACM.

Sandeep Kumar Shukla is an associate professor in the Bradley Department of Electrical and Computer Engineering at Virginia Polytechnic and State University. He is the founder and director of the FERMAT lab, and cofounder and deputy director of the Center for Embedded Systems for Critical Applications (CESCA). He has a PhD in computer science from the State University of New York at Albany. He is a senior member of the IEEE and ACM.

Janos Sztipanovits is the E. Bronson Ingram Distinguished Professor of Engineering at Vanderbilt University and founding director of the Institute for Software Integrated Systems (ISIS) there. He has a PhD in electrical engineering from the Technical University of Budapest. He is a Fellow of the IEEE.

Guang Yang is a staff software research and development engineer in the National Instruments Berkeley, California, office. He has a PhD in electronics engineering and computer sciences from the University of California, Berkeley. He is a member of the IEEE.

Deepak A. Mathaikutty is a researcher at the Microarchitecture Research Lab at Intel. He has a PhD in computer engineering from Virginia Polytechnic and State University. He is a member of the IEEE.

■ Direct questions and comments about this article to Sandeep K. Shukla, 302 Whittemore Hall, Virginia Tech, Blacksburg, VA 24061; shukla@vt.edu.