COVER FEATURE

Developing Applications Using Model-Driven Design Environments

Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema
Vanderbilt University

Model-driven development is an emerging paradigm that improves the software development life cycle, particularly for large software systems, by providing a higher level of abstraction for system design than is possible with third-generation programming languages.

Historically, software development methodologies have focused more on improving tools for system development than on developing tools that assist with system composition and integration. Component-based middleware like Enterprise JavaBeans (EJB), Microsoft .NET, and the CORBA Component Model (CCM) have helped improve software reusability through component abstraction. However, as developers have adopted these commercial off-the-shelf technologies, a wide gap has emerged between the availability and sophistication of standard software development tools like compilers and debuggers, and the tools that developers use to compose, analyze, and test a complete system or system of systems. As a result, developers continue to accomplish system integration using ad hoc methods without the support of automated tools.

Model-driven development is an emerging paradigm that solves numerous problems associated with the composition and integration of large-scale systems while leveraging advances in software development technologies such as component-based middleware. MDD elevates software development to a higher level of abstraction than is possible with third-generation programming languages.

MODEL-DRIVEN DEVELOPMENT

MDD uses models to represent a system's elements and their relationships. Models serve as input and output at all stages of system development until the final system is generated. A popular variant of MDD is the Object Management Group's Model-Driven Architecture (MDA),[1] which represents systems using OMG's general-purpose Unified Modeling Language (along with specific profiles) and transforms these models into artifacts that run on a variety of platforms like EJB, .NET, and CCM.

Unlike MDA, Model-Integrated Computing (MIC),[2] a variant of MDD, uses domain-specific modeling languages (DSMLs) to represent system elements and their relationships as well as their transformations to platform-specific artifacts. We have successfully applied the MIC concept to develop several DSML tool suites. Here, we focus on two:

- Platform-Independent Component Modeling Language (PICML), which assists in developing, configuring, and deploying systems using component middleware technology such as CCM; and
- Embedded Control Systems Language (ECSL), which supports development of distributed embedded automotive applications.

Both of these tool suites are built using the Generic Modeling Environment (GME),[3] an open source, metaprogrammable, domain-specific design environment that developers use to create both DSMLs and models that conform to these DSMLs within the same graphical environment.

Generic Modeling Environment

GME is an open source, visual, configurable design environment for creating domain-specific modeling languages (DSMLs) and program synthesis environments (http://escher.isis.vanderbilt.edu). One of GME's unique features is that it is metaprogrammable—that is, GME is used not only to build DSMLs but also to build models that conform to a DSML. In fact, *MetaGME*, the GME environment used to build DSMLs, is itself built using another DSML—or metametamodel. GME provides the following elements to define a DSML:

- *project:* the top-level container in a DSML;
- *folders*: used to group collections of similar elements together;
- *atoms*: the indivisible elements of a DSML, used to represent the leaf-level elements in a DSML;
- *models*: the compound objects in a DSML, used to contain different types of elements—also known as parts—like references, sets, atoms, and connections;
- *aspects*: used primarily to provide a different viewpoint of the same model (every part of a model is associated with an aspect);
- *connections*: used to represent relationships between domain elements;
- *references*: used to refer to other elements in different portions of a DSML hierarchy (unlike connections, which can be used to connect elements within a model); and
- *sets*: containers whose elements are defined within the same aspect and have the same container as the owner.

DOMAIN-SPECIFIC MODELING LANGUAGES

DSMLs are the backbone of model-integrated computing. A DSML can range from something that is very specific, such as the elements of a radar system, to something as broad as a component-based middleware application built using platforms such as EJB or CCM. The key idea behind a DSML is its ability to capture domain elements as first-class objects. A DSML can be viewed as a five tuple:[4]

- a *concrete syntax* defining the specific notation (textual or graphical) used to express domain elements;
- an *abstract syntax* defining the concepts, relationships, and integrity constraints available in the language;
- a *semantic domain* defining the formalism used to map the semantics of the models to a particular domain;
- a *syntactic mapping* assigning syntactic constructs (graphical or textual) to elements of the abstract syntax; and
- a *semantic mapping* relating the syntactic concepts to the semantic domain.

Thus, the abstract syntax determines all syntactically correct sentences—or models—in the language. A DSML is also known as a *metamodel*, because a DSML is itself a model that defines all the possible models that developers can build using it.

CREATING A DSML

DSMLs are defined visually in GME. The first step is to identify the different domain elements and their relationships that we want to model, which is usually done with iterative input from a domain expert. Any MDD tool infrastructure should allow modification of the elements (or relationships among elements) with ease.

Second, we map the domain elements to GME concepts like models and atoms, as described in the "Generic Modeling Environment" sidebar. This process is similar to defining types in a programming language like C/C++. To capture the DSML's concrete syntax, developers define the types in the DSML.

With the mapping of domain elements to GME concepts, UML class diagrams represent the DSML elements as shown in Figure 1. With GME, users can customize the visualization of DSML elements—that is, its concrete syntax—using a *decorator*, a component written in a traditional programming language that implements a set of standard callback interfaces. Once a decorator is registered with GME, the environment invokes the callbacks whenever it needs to display the element.

Using UML class diagrams as the notation for a DSML's concrete syntax, developers can capture some association between elements because semantics like cardinality can be sufficiently represented in UML. To further constrain associations, GME uses OMG's Object Constraint Language[5] to provide a built-in constraint manager, as Figure 2 shows. Since they do not take system dynamics into account, DSML semantics using OCL constraints are static.

After defining the DSML's elements and its associated static semantics, the developer instructs GME to generate a customized DSML environment using a process called *metainterpretation*. This process takes the definition of the DSML from the previous step; runs a set of standard transformations ensuring consistency (as does a traditional compiler); and creates a paradigm file defining the DSML, which is then registered with GME. It is now possible to create models conforming to the DSML using GME.

Although we have defined the elements, the relationships, and the DSML's static semantics, we also must define dynamic semantics to make the DSML useful for complex real-world applications. A DSML's dynamic semantics can be enforced by applying model interpreters written using a traditional programming language like C++, Python, or Java and registered with GME. When the interpreter is invoked, GME gives it access to the model hierarchy, and it can perform different kinds of validation and generative operations on the models. One such operation can include generating platform-specific artifacts directly from the models.

APPLYING MIC TO COMPONENT-BASED APPLICATIONS

A common trend in component middleware technology is the use of metadata to capture application properties that were previously tightly coupled with the implementation. This allows for declarative specification of an application using platform-agnostic technologies like XML and then automatic deployment and configuration.

Higher-level abstractions like virtual machines, execution containers, and extra information about systems via rich metadata has a direct impact in enabling people to build systems that are more heterogeneous than previously possible. However, this also increases the amount of information that must be managed by the system developer as well as the complexity of system integration.

Infrastructure for managing such complex deployment was essentially lacking in previous-generation middleware, and most deployments were done on an ad hoc basis without much infrastructure reuse or potential for rigorous system verification and validation. This lack of simplification and automation has hindered the adoption—and ostensible benefits—of component middleware technologies.

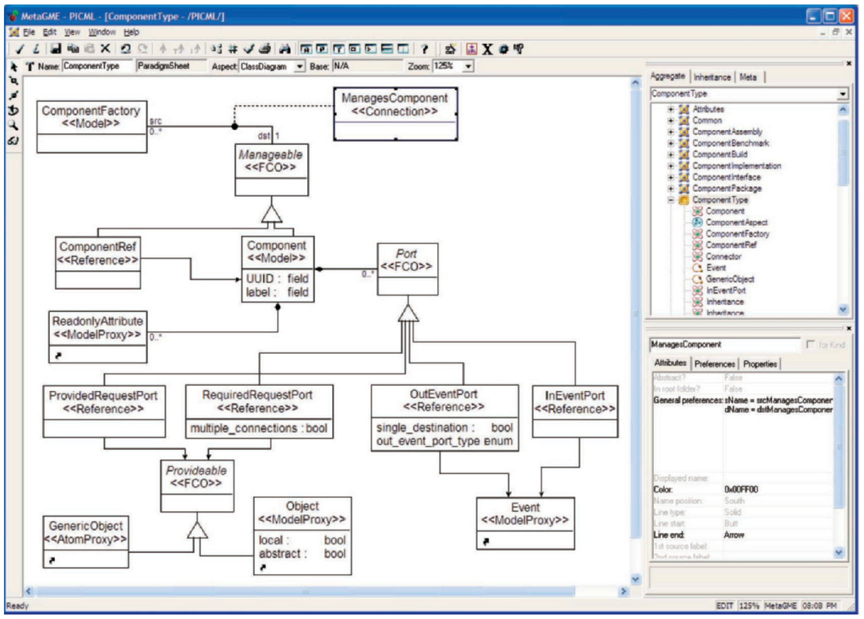
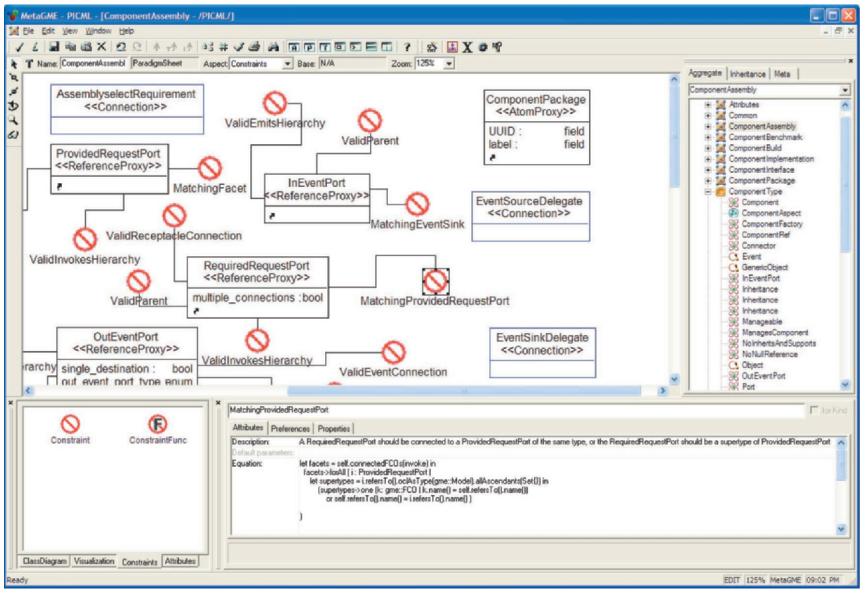

MetaGME - PICML - [ComponentType - /PICML/]
ComponentFactory <<Model>>
ManagesComponent <<Connection>>
Manageable <<FCO>>
ComponentRef <<Reference>>
Component <<Model>>
UUID : field
label : field
Port <<FCO>>
ReadonlyAttribute <<ModelProxy>>
ProvidedRequestPort <<Reference>>
RequiredRequestPort <<Reference>>
multiple_connections : bool
OutEventPort <<Reference>>
single_destination : bool
out_event_port_type enum
InEventPort <<Reference>>
Provideable <<FCO>>
GenericObject <<AtomProxy>>
Object <<ModelProxy>>
local : bool
abstract : bool
Event <<ModelProxy>>
Aggregate
Inheritance
Meta

Figure 1. Domain mapping. With the Generic Modeling Environment, users can customize a domain-specific modeling language's concrete syntax.


MetaGME - PICML - [ComponentAssembly - /PICML/]
AssemblyselectRequirement <<Connection>>
ValidEmitsHierarchy
ValidParent
ComponentPackage <<AtomProxy>>
ProvidedRequestPort <<ReferenceProxy>>
MatchingFacet
InEventPort <<ReferenceProxy>>
MatchingEventSink
EventSourceDelegate <<Connection>>
ValidReceptacleConnection
ValidInvokesHierarchy
RequiredRequestPort <<ReferenceProxy>>
multiple_connections : bool
MatchingProvidedRequestPort
OutEventPort <<ReferenceProxy>>
single_destination : bool
ValidInvokesHierarchy
ValidEventConnection
EventSinkDelegate <<Connection>>
Constraint
ConstraintFunc
ClassDiagram
Visualization
Constraints
Attributes

MatchingProvidedRequestPort

Description: A RequiredRequestPort should be connected to a ProvidedRequestPort of the same type, or the RequiredRequestPort should be a supertype of ProvidedRequestPort

```
Equation:  let facets = self.connectedFCOs(invoke) in
            facets->forAll ( i : ProvidedRequestPort |
              let supertypes = i.refersTo().oclAsType(gme::Model).allAscendants(Set{}) in
                (supertypes->one (k: gme::FCO | k.name() = self.refersTo().name())
                  or self.refersTo().name() = i.refersTo().name() )

            )
```

Figure 2. Defining static semantics in GME. To further constrain associations, GME uses OMG's Object Constraint Language to provide a built-in constraint manager.

PICML

To address these problems, we developed PICML, an open source DSML available for download as part of the CoSMIC MDD framework (www.dre.vanderbilt.edu/CoSMIC). Developers of component-based systems can use PICML to define application interfaces, quality-of-service (QoS) parameters, and system-software building rules, as well as to generate metadata and XML descriptor files that enable automated system deployment. PICML also provides capabilities to handle complex component engineering tasks, such as multi-aspect visualization of components and subsystem interactions, component deployment planning, and hierarchical modeling of component assemblies.

Currently, PICML is used in conjunction with the Component-Integrated ACE ORB (CIAO)—our CCM implementation—and our Deployment and Configuration Engine (DAnCE),[6] a QoS-enabled deployment

Deployment Metadata

PICML generates the following types of deployment descriptors based on the OMG Deployment and Configuration specification:

- *component interface descriptor* (.ccd): describes a single component's interfaces, ports, and attributes;
- *implementation artifact descriptor* (.iad): describes a single component's implementation artifacts—for example, DLLs and executables;
- *component implementation descriptor* (.cid): describes a specific implementation of a component interface and also contains component interconnection information;
- *component package descriptor* (.cpd): describes a single component's multiple alternative implementations—for example, different operating systems;
- *package configuration descriptor* (.pcd): describes a component package configured for a particular requirement;
- *component deployment plan* (.cdp): describes the plan that guides the runtime deployment; and
- *component domain descriptor* (.cdd): describes the deployment target—the nodes and networks—on which the components are to be deployed.

engine. However, PICML's design has been driven by the goal of integrating systems built using different component technologies like .NET and EJB.

PICML is defined as a metamodel in GME for describing components, types of allowed interconnections between components, and types of component metadata for deployment. From this metamodel, the metamodel interpreter generates 20,000 lines of C++ code representing the modeling language elements as equivalent C++ types. This generated code allows manipulation of modeling elements—that is, instances of C++ language types—and forms the basis for writing model interpreters, which traverse the model hierarchy to generate XML-based deployment descriptors. As the "Deployment Metadata" sidebar describes, these descriptors support OMG's Deployment and Configuration specification.[7]

Figure 3 shows the typical steps involved in component-based application development using PICML's MDD approach:

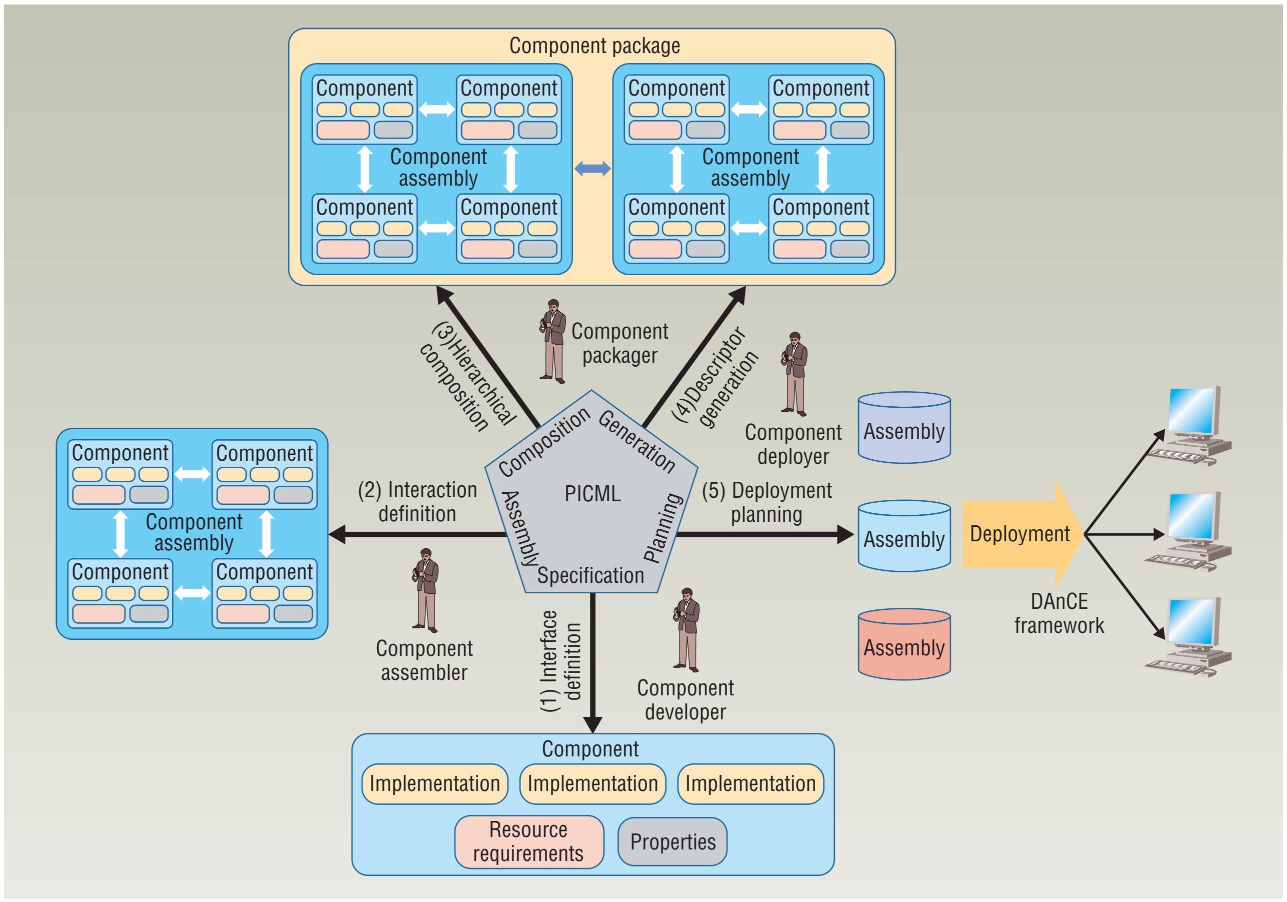


Figure 3. Model-driven development of component-based applications using PICML. The infrastructure uses artifacts output from PICML to carry out the different stages of component application deployment.

1. *Visual component interface definition*. Using PICML, we create a set of component interface and other data type definitions in CORBA's Interface Definition Language. In this step, we can either import existing IDL definitions or create new ones from scratch.

To import existing definitions, the IDL importer can easily migrate existing CORBA to PICML. The importer takes IDL files as input, maps their contents to the appropriate PICML model elements, and generates a single XML file that can be imported as a PICML model. To design new interfaces, PICML's graphical modeling environment supports using an intuitive drag-and-drop technique, making this process largely self-explanatory and independent of platform-specific technical knowledge.

2. *Valid component interaction definition*. By elevating the level of abstraction, the well-formedness rules of DSMLs like PICML actually capture semantic information such as constraints on model composition and allowed interactions.

PICML defines a hierarchy that developers can use to view their system at multiple levels of detail depending upon their needs.

There is a significant difference in the early detection of errors in the MDD paradigm compared with traditional object-oriented or procedural development using a conventional compiler. PICML uses OCL constraints to define the modeling language's static semantics, thereby disallowing invalid systems. In other words, PICML enforces the correct-by-construction approach to system development.

3. *Hierarchical composition*. In a complex system with thousands of components, visualization becomes an issue because of the practical limitations of displays as well as human cognition. Without support for hierarchical composition, observing and understanding system representations in a visual medium does not scale.

To increase scalability, PICML defines a hierarchy that developers can use to view their system at multiple levels of detail depending upon their needs. With hierarchical composition, developers can both visualize their systems and compose systems from smaller subsystems. This feature supports unlimited levels of hierarchy—constrained only by the physical memory of the system used to build models—and promotes the reuse of component assemblies. Therefore, developers can use PICML to develop repositories of predefined components and subsystems.

The hierarchical composition capabilities that PICML provides are only a logical abstraction: Deployment plans generated from PICML flatten out the hierarchy to connect the two destination ports directly, thereby ensuring that no extra communication overhead is attributed to this abstraction at runtime.

4. *Valid deployment descriptor generation*. In addition to ensuring design-time integrity of systems built using OCL constraints, PICML also generates the complete set of deployment descriptors needed as input to the component deployment mechanisms. These descriptors conform to the OMG D&C specification[7] and are described in the "Deployment Metadata" sidebar.

5. *Deployment planning*. Systems are often deployed in heterogeneous execution environments. To support these needs, PICML can specify the target environment, which includes nodes, interconnects among nodes, and bridges among interconnects. Once the target environment is specified, developers can allocate component instances onto the target environment's various nodes. PICML currently provides facilities for specifying static allocation of components to nodes.

EMBEDDED DESIGN USING ECSL

The model-based approach for embedded systems development has heretofore been confined to the functional aspects of the system design and is restricted to a limited suite of tools—most notably the Mathworks family of Matlab, Simulink, and Stateflow tools.

Simulink and Stateflow are powerful graphical system design tools for modeling and simulating continuous and discrete event-based behavior in a dynamic system. However, these tools by no means cover the entire spectrum of embedded systems development.

The embedded systems development process includes several other complex activities such as requirements specification, verification, mapping onto a distributed platform, scheduling, performance analysis, and synthesis. Although some existing tools individually support one or more of these development activities, integration with the Mathworks suite is often lacking. This makes it difficult to maintain a consistent view of the system as the design progresses and requires significant manual effort to create different representations of the same system.

To address these deficiencies, our Embedded Control Systems Language (ECSL) supports

- annotation of structural design, software-component design, and behavior implementation to supply information needed by a code generator;
- creation of hardware-topology design models, electronic control unit (ECU)-design models, and firmware-implementation design models; and
- creation of deployment models that capture component and communication mapping.

ECSL also imports existing Simulink/Stateflow (SL/SF) models into the GME environment.

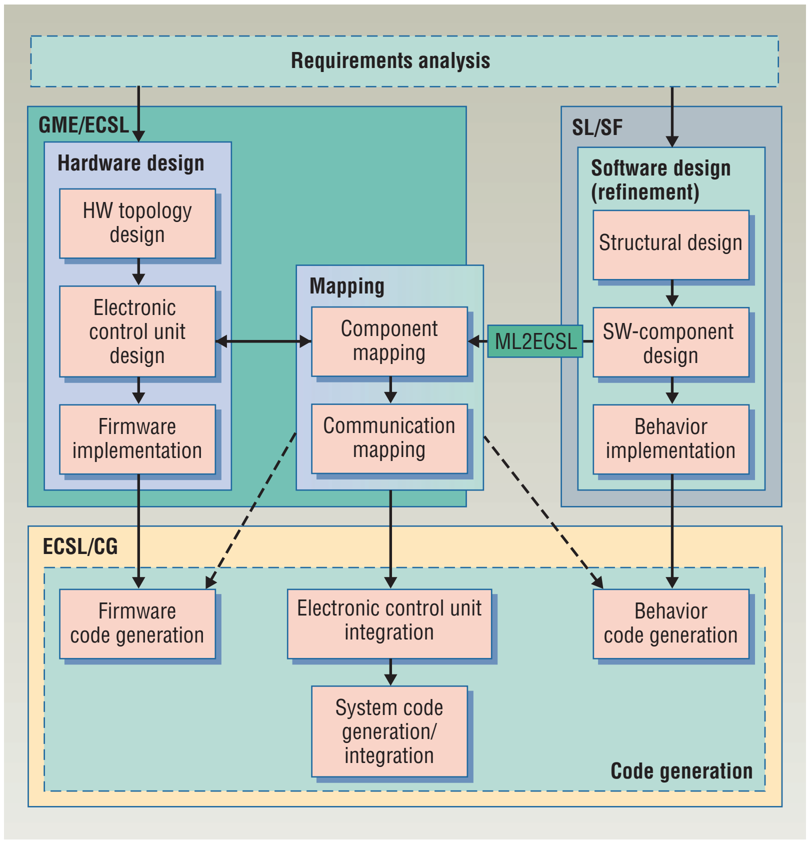


Figure 4. ECSL process. Each rectangular block denotes a particular activity, and arrows indicate the workflow between different activities. Activities can roughly be grouped in three blocks: hardware design, software design, and mapping.

Embedded automotive applications

Designing and developing embedded automotive systems is becoming notoriously difficult. In recent years, there has been an explosion in the scale and complexity of these systems, with a push toward drive-by-wire technologies, increasing feature levels, and increasing capabilities in embedded computing platforms. To address this level of complexity, the automotive industry has in general embraced the model-based approach for embedded systems development.

Figure 4 depicts the activities in an automotive embedded systems development process as supported by the ECSL tools. Each rectangular block denotes a particular activity, and arrows indicate the workflow between different activities. Activities can roughly be grouped in three blocks: hardware design, software design, and mapping. (Requirements engineering is not within the scope of this tool suite, although it is the basis for most of the modeling and development activities.)

Hardware design includes specifying ECUs in a network and their connections with buses and defining an architectural topology of the distributed embedded platform. Refinements of this activity include designing individual ECUs, selecting the processors, and determining the memory and I/O requirements.

Software design includes

- *structural design:* the hierarchical decomposition of the embedded system into subsystems from a functional viewpoint;
- *component design*: another form of decomposition, not independent of the functional decomposition, dealing with more classical embedded software concerns such as real-time requirements, real-time tasks, periodicity, deadlines, and scheduling; and
- *functional/behavioral design*: the elaboration of the hierarchical structural design's leaf elements in terms of a synthesizable realization.

Mapping includes activities involving both software and hardware objects, such as decisions regarding the deployment of certain complete or partial functions to hardware nodes that are part of the network and the assignment of signals to bus messages.

The abstract design process shown in Figure 4 is made concrete by a number of supporting tools, including GME/ECSL, ML2ECSL, and ECSL/CG. GME/ECSL is the Generic Modeling Environment tailored to support the ECSL modeling language. The ML2ECSL translator imports SL/SF models into the ECSL environment. Finally, ECSL/CG is a specialized code generator that produces various low-level production artifacts from ECSL models, including real-time operating system configuration, firmware configuration code, and behavioral implementation of the components.

ECSL concepts

ECSL, a graphical modeling language built using GME that contains modeling concepts for design activities, comprises a suite of sublanguages.

Functional modeling. ECSL's SL/SF sublanguages mirror the capabilities found in the Simulink and Stateflow languages such that all SL/SF models can be imported into the GME/ECSL environment. Simulink follows a dataflow-diagram visual notation, while Stateflow supports Statechart-like hierarchical finite state machines.

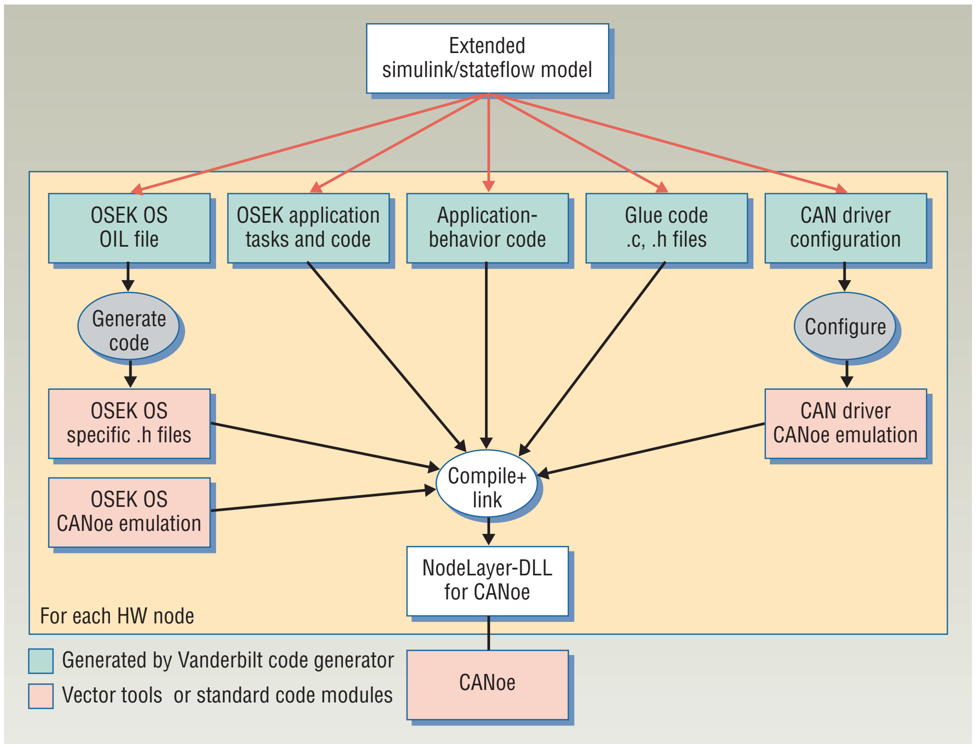
Component modeling. This sublanguage allows software modeling in two stages: integrating and then componentizing SL/SF models. Componentization is specified using the GME containment and reference capabilities: Components are GME models that contain references to elements of the functional model imported from SL/SF.

Components consist of ports that allow the specification of intercomponent communication, as well as interfaces to physical devices including sensors and actuators. Port attributes capture required communication properties such as data type, scaling, and bit width. The intracomponent dataflow exists within the imported functional models. The designer introduces the intercomponent dataflow after componentizing the imported models.

Hardware topology modeling. Designers can use ECSL's hardware modeling to specify the hardware topology, including the processors and communication links between the processors. ECU models represent specific processors and have two ports representing the I/O channels and the bus connections. ECU firmware specifics with respect to memory size and CPU speed are captured with attributes. I/O channel ports come in two variants: sensor ports and actuator ports. Bus models represent communication pathways used to connect ECUs and are expressed as GME atoms whose attributes specify various physical communication system properties such as bit rates.

Deployment (mapping) modeling. This modeling sublanguage captures how software components are deployed on the hardware. The designer can use the ECU model's *deployment aspect* to map the software component to the ECU using GME's reference concept.

Note that deployment models are separate from software models, thus allowing the reuse of software models in different hardware architectures. Furthermore, component ports are connected to ECU ports to indicate how the component software interfaces map to actual sensors, actuators, and buses.



Extended simulink/stateflow model
OSEK OS OIL file
OSEK application tasks and code
Application-behavior code
Glue code .c, .h files
CAN driver configuration
Generate code
Configure
OSEK OS specific .h files
CAN driver CANoe emulation
OSEK OS CANoe emulation
Compile+ link
NodeLayer-DLL for CANoe
For each HW node
CANoe
Generated by Vanderbilt code generator
Vector tools or standard code modules

Figure 5. ECSL code generation. The ECSL/CG tool produces code artifacts necessary for system implementation. CAN = controller area network.

Generating code and other artifacts

As Figure 5 shows, the ECSL/CG tool is a code generator that produces code artifacts necessary for system implementation. The tool generates the following types of files:

- an OIL (OSEK Implementation Language) file for each ECU node that includes a listing of all used OSEK (www.osek-vdx.org) objects and their relations;
- one or more C files implementing OSEK tasks;
- application-behavior code called out from within a task frame; and
- glue code comprising one or more C code/header files that resolve the calls to the controller area network (CAN) driver or the firmware to provide access to CAN signals or hardware I/O signals.

Together, these generated artifacts comprise the entire application code. While the code generator supports synthesis of application-behavior code, it is feasible to link in externally supplied application-behavior code, which can be either handwritten or generated using a third-party tool. Additional framework code is generated using vendor-supplied tools such as those from Vektor Informatik that configure the controller area network bus. Also, developers can use the OSEK vendor-supplied tools to process the generated OIL files that configure the operating system and set up the tasks and task properties.

We have evaluated the code generator with several examples of small to medium complexity such as a defroster controller and a steering and braking controller. The generated code for these examples was functionally correct and bug-free. We can systematically derive the correctness based on the glue code construction; however, a formal proof of correctness of the application-behavior code generation is an open problem and

remains a work in progress. We are also investigating techniques for automatically generating a test suite for the application behavior code generation.

The automotive application domain that the tools target emphasizes the readability and traceability of generated code. Therefore, the code generator does not perform any aggressive optimization on the Statecharts specifying application component behavior.

Model-driven development is a promising paradigm for tackling system composition and integration challenges. MDD elevates the abstraction level of software development and bridges the gap between technology domains by allowing domain experts (who may not be experts in software development) to design and build systems. This higher level of abstraction also allows transformation between models in different domains without resorting to low-level integration solutions such as standard network protocols.

The Platform-Independent Component Modeling Language, a DSML built using GME, simplifies and automates many activities associated with developing and deploying component-based systems. The Embedded Control Systems Language is a DSML tool suite that supports development of distributed embedded automotive systems by interfacing with existing engineering tools, while adding new capabilities to support deployment on distributed platforms and integration of model verification tools into the development process.

MDD ensures the semantic consistency of systems by enforcing the correct-by-construction philosophy. It also solves many of the accidental complexities that arise during system integration due to the heterogeneity of the underlying component middleware technologies. With improvements in generative techniques, MDD has the potential to automate code generation just as compilers replaced assembly language/machine-code programming. ■

References

1. D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.
2. J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *Computer*, Apr. 1997, pp. 110-112.
3. A. Ledeczi et al., "Composing Domain-Specific Design Environments," *Computer*, Nov. 2001, pp. 44-51.
4. G. Karsai et al., "Model-Integrated Development of Embedded Software," *Proc. IEEE*, Jan. 2003, pp. 145-164.
5. Object Management Group, *Unified Modeling Language: OCL version 2.0*, OMG document ptc/03-10-14 ed., Oct. 2003.
6. G. Deng et al., "DAnCE: A QoS-Enabled Component Deployment and Configuration Engine," *Proc. 3rd Working Conf. Component Deployment*, LNCS 3798, Springer-Verlag, 2005, pp. 67-82.
7. Object Management Group, *Deployment and Configuration Adopted Submission*, OMG document ptc/03-07-08 ed., July 2003.

Krishnakumar Balasubramanian *is a graduate student in the Department of Electrical Engineering and Computer Science at Vanderbilt University. His research interests include model-driven development, deployment and configuration of component middleware, and patterns and frameworks for distributed, real-time, and embedded systems development. He received an MS in computer science from Washington University in St. Louis. He is a student member of the IEEE. Contact him at kitty@dre.vanderbilt.edu.*

Aniruddha Gokhale *is an assistant professor of electrical engineering and computer science at Vanderbilt University and a senior research scientist in the Institute for Software-Integrated Systems at Vanderbilt. His research interests include real-time component middleware optimizations, model-driven software development, and distributed resource management. Gokhale received a PhD in computer science from Washington University in St. Louis. He is a member of the IEEE. Contact him at a.gokhale@vanderbilt.edu.*

Gabor Karsai *is an associate professor of electrical engineering and computer science at Vanderbilt University and a senior research scientist in the Institute for Software-Integrated Systems at Vanderbilt. He conducts research in model-integrated computing. Karsai received a technical doctorate degree in electrical engineering from the Technical University of Budapest, Hungary, and a PhD in electrical engineering from Vanderbilt University. He is a member of the IEEE Computer Society. Contact him at gabor.karsai@vanderbilt.edu.*

Janos Sztipanovits *is the E. Bronson Ingram Distinguished Professor of Engineering in the Department of Electrical Engineering and Computer Science at Vanderbilt University. He is founding director of the Institute for Software-Integrated Systems. His research interests include model-integrated computing, structurally adaptive systems, and embedded software and systems. Sztipanovits received a distinguished doctor degree from the Hungarian Academy of Sciences. He is a Fellow of the IEEE. Contact him at janos.sztipanovits@vanderbilt.edu.*

Sandeep Neema *is a research assistant professor in the Institute for Software-Integrated Systems at Vanderbilt University. His research interests include design space exploration and constraint-based synthesis of parallel/distributed, real-time, embedded systems. Neema received a PhD in electrical and computer engineering from Vanderbilt University. He is a member of the IEEE. Contact him at sandeep.k.neema@vanderbilt.edu.*