Developing Applications Using Model-driven Design Environments

Krishnakumar Balasubramanian, *Member, IEEE,* Aniruddha Gokhale, *Member, IEEE,* Gabor Karsai, *Senior Member, IEEE,* Janos Sztipanovits, *Fellow, IEEE,* and Sandeep Neema, *Member, IEEE*

***Abstract*— Model-driven development (MDD) is an emerging paradigm that improves the software development lifecycle, particularly for large software systems by providing a higher-level of abstraction for designing and developing the system than is possible with third-generation programming languages. The MDD paradigm relies on the use of (1) Domain-Specific Modeling Languages that incorporate elements of the domain being modeled and their relationship as first-class objects, and (2) model transformations that transform the models into platform-specific artifacts, such as code. This paper illustrates several key characteristics of the MDD approach that differentiate it from traditional software development approaches. Additionally, the paper describes meta-programmable tools used to construct domain-specific tool-suites, and provides two example tool suites drawn from different domains. These tool-suites are (a) PICML, which supports the development of standards-compliant component-based applications, and (b) ECSL, which supports software development for distributed embedded controllers.**

***Index Terms*— Model-Integrated Computing, GME, CCM, CoSMIC, Deployment & Configuration**

I. Introduction

HISTORICALLY, software development methodologies have focused on improving the individual tools like programming languages, much more than tools that assist system composition and system integration. Component-based middleware like Enterprise Java Beans, Microsoft .NET and the CORBA Component Model (CCM) have helped improve the re-usability of software through the abstraction of components. However, as systems are being built more frequently using commercial-off-the-shelf (COTS) technologies, a wide gap has been created in the availability and level of sophistication of tools that help with software development like compilers, debuggers, linkers and virtual machines, as opposed to tools that allow developers to compose, analyze and test a complete system, or system of systems. As a result, the task of system integration continues to be accomplished using *ad hoc* methods without the support of automated tools.

Model-Driven Development (MDD) is an emerging paradigm which solves a number of problems associated with the composition and integration of large-scale systems while also leveraging the advances in software development, such as component-based middleware. The focus of MDD is to elevate software development to a higher level of abstraction than that provided by third generation programming languages. The MDD approach relies on the use of models to represent the system elements of the domain and their relationships. In MDD, models are used in most system development activities, *i.e.*, models serve as input and output at all stages of system development until the final system is itself generated.

Contact Author e-mail: kitty@dre.vanderbilt.edu
The authors are with Vanderbilt University, Nashville, TN, USA.

This paper describes one variant of MDD called Model-Integrated Computing (MIC) [1], which focuses on using Domain-Specific Modeling Languages (DSMLs) to represent system elements and their relationships, and transformations from these languages to platform-specific artifacts. We have successfully applied the concept of MIC to develop a number of DSML tool-suites. In this paper we will focus on two such tool suites: (1) **Platform-Independent Component Modeling Language (PICML)**, which assists developers in the development, configuration and deployment of systems built using component middleware technology, such as CCM; (2) **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) [2], an open-source *meta-programmable* domain-specific design environment, which allows development of both DSMLs and the models that conform to these DSMLs, within the same graphical environment.

Another popular variant of MDD is the Object Management Group's Model-Driven Architecture (MDA) [3], which relies on representing systems using the Unified Modeling Language (UML), a general purpose modeling language, and transforming these models into artifacts that run on a variety of platforms like EJB, CCM and .NET. Unlike MDA, which focuses on the use of a general-purpose modeling language like UML (along with specific profiles), MIC focuses on developing modeling languages that are tailored to a particular domain of interest.

The remainder of the paper is organized as follows: Section II describes the concept of DSML and the key elements defined in it; Section III describes PICML and shows how PICML addresses the challenges involved in developing component-based applications; Section IV describes ECSL and shows how ECSL resolves the challenges in developing embedded automotive applications; Section V compares our work with other MDD frameworks and tools; and finally Section VI presents concluding remarks.

II. Domain-Specific Modeling Languages

DSMLs are the backbone for realizing the vision of Model-Integrated Computing (MIC), a variant of MDD. One difference between other variants of MDD like MDA and MIC is

the emphasis on the "domain-specific" aspect of the modeling languages. This section provides an overview of DSMLs and explains the steps needed to create a DSML.

A. Overview of DSML

The domain of interest of a DSML can range from something very specific, such as the elements of a radar system, or can be as broad as the set of all the component-based middleware applications built using platforms such as EJB, or CCM. The key idea behind DSMLs is their ability to capture the elements of the domain as *first-class objects*.

A DSML can be viewed as a five-tuple [4] consisting of:

1) **Concrete Syntax (C)**, which defines the specific notation (textual or graphical) used to express domain elements,
2) **Abstract Syntax (A)**, which defines the concepts, relationships and integrity constraints available in the language,
3) **Semantic Domain (S)**, which defines the formalism used to map the semantics of the models to a particular domain,
4) **Syntactic Mapping ($\mathbf{M}_C$: A→C)**, which assigns syntactic constructs (*e.g.*, graphical and/or textual) to elements of the abstract syntax, and
5) **Semantic Mapping ($\mathbf{M}_S$: A→S)**, which relates the syntactic concepts to those of the semantic domain.

Thus the abstract syntax determines all the (syntactically) correct "sentences", *i.e.*, models, in the language. A DSML is also known as a "metamodel", since a DSML is (itself) a model that defines all the possible models that can be built using it. To support the development of DSMLs, we have developed a "meta-programmable" modeling environment called the Generic Modeling Environment (GME).

B. Creating a DSML using GME

DSMLs are defined visually in GME. The following are the typical steps in creating a DSML using GME:

1. Identifying the domain elements and their relationships. The first step in creating a DSML is to identify the different elements of the domain that we want to model, and their relationships. This is usually done with input from a domain expert. It is unlikely that all the elements are defined in a single iteration. Just like software development, MDD is also an iterative process, and any MDD tool infrastructure should allow modification of the elements (or relationships between elements) with ease.

2. Mapping the elements of the domain to concepts in GME. Sidebar 1 provides more information about the GME concepts. Once we have enumerated the elements of the domain, we need to map these domain concepts to GME concepts like Models, Atoms, *et al*. This process is very similar to defining the types in a program when coding using a language like C/C++.

3. Using GME concrete syntax to realize the mapping. With the mapping of elements of the domain to concepts in GME complete, the next step is to use the concrete syntax of GME, *i.e.*, UML class diagrams, to capture the elements of the DSML as shown in Figure 1. It is important to note that the concrete syntax of the elements of your DSML, *i.e.*, visualization of elements of the DSML, is defined when you are defining the types of your your DSML. GME provides the ability to customize the concrete syntax, *i.e.*, customize the visualization of elements in your DSML. Customization of visualization is done using *decorators* in GME. A *decorator* is a component (written using a traditional programming language) that implements a set of standard callback interfaces. Once a decorator is registered with GME, GME invokes the callbacks whenever GME needs to display the element.

4. Defining static semantics of the DSML. The use of UML class diagrams as the notation for concrete syntax of DSMLs in GME allows capturing some semantics of the association between the different elements. This is because some semantics like cardinality of associations can be sufficiently represented using the notations offered by UML class diagrams. For constraining the associations between elements further, GME provides a built-in constraint manager. GME's constraint manager allows the definition of constraints using OMG's Object Constraint Language (OCL) [5] as shown in Figure 2. The semantics of the DSML that are enforceable using OCL can be viewed as *static semantics*, since they do not take the system dynamics into account.

Sidebar 1: Generic Modeling Environment

Generic Modeling Environment (GME) is an open-source, visual, configurable design environment for creating DSMLs and program synthesis environments, available for download from `escher.isis.vanderbilt.edu/downloads?tool=GME`. A unique feature of GME is that it is a *meta-programmable* environment. By *meta-programmable*, we refer to the fact that GME is not only used to build DSMLs, but also to build models that conform to a DSML. In fact, the environment used to build DSMLs in GME is itself built using another DSML (also known as the *meta-metamodel*) called "MetaGME". GME provides the following elements to define a DSML:

- **Project**, which is the top-level container in a DSML,
- **Folders**, which are used to group collections of similar elements together,
- **Atoms**, which are the indivisible elements of a DSML, and used to represent the leaf-level elements in a DSML,
- **Models**, which are the compound objects in a DSML, and are used to contain different types of elements like References, Sets, Atoms, Connections *et al.* (the elements that are contained by a Model are known as *parts*),
- **Aspects**, which are primarily used to provide a different viewpoint of the same Model (every part of a Model is associated with an Aspect),
- **Connections**, which are used to represent relationships between the elements of the domain,
- **References**, which are 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),
- **Sets**, which are containers whose contained elements are defined within the same aspect and have the same container as the owner.

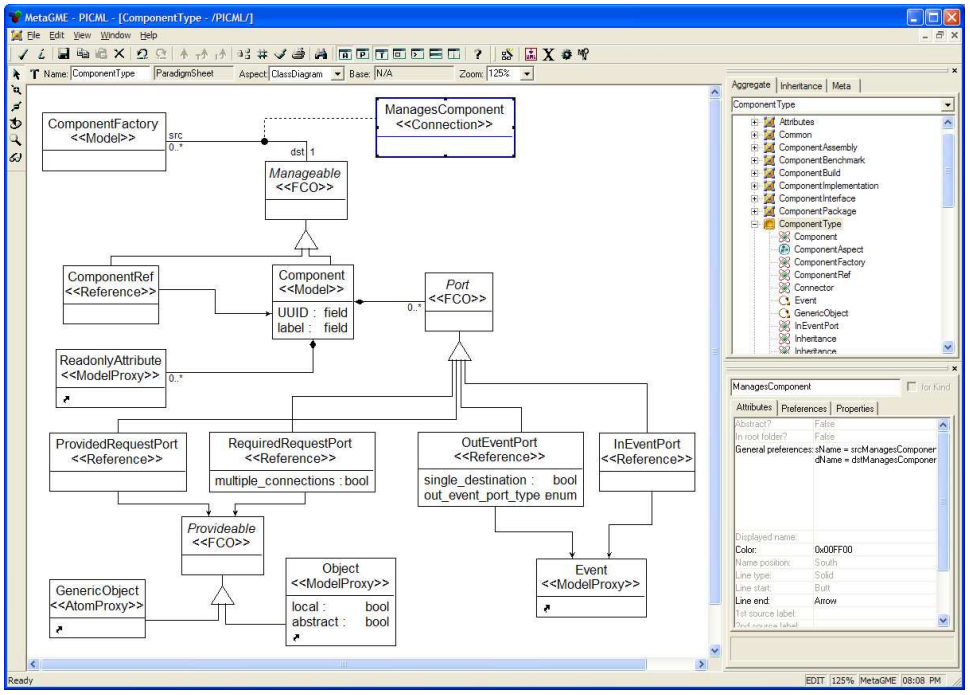

Fig. 1: Realizing the domain mapping using concrete syntax in GME

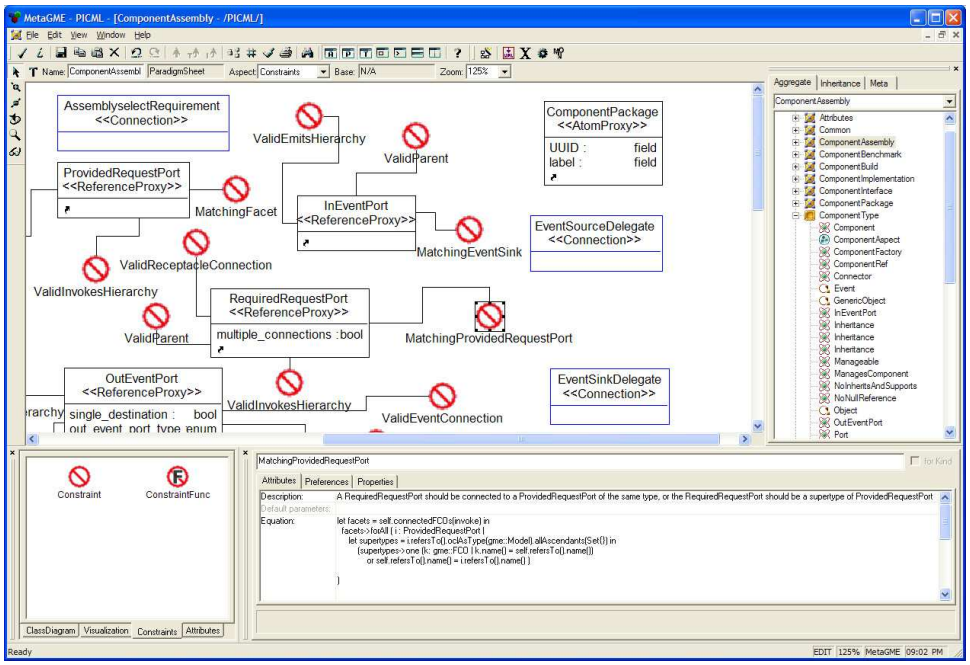

Fig. 2: Defining static semantics in GME

5. Generating the DSML environment. Now that we have defined the elements of the DSML and the associated *static semantics*, we instruct GME to generate our customized DSML environment. This is done using a process called *meta-interpretation*. Meta-interpretation takes the definition of the DSML from the previous step, runs a set of standard transformations (which ensure consistency of the language) just like a traditional compiler and creates a *paradigm*. The paradigm file, which actually defines the DSML, is then registered with the GME environment. It is now possible to create models that conform to the DSML that we have built using GME.

6. Defining dynamic semantics of the DSML. Though we have defined the elements, relationships and the static semantics of the DSML, it is necessary to define *dynamic semantics* to make the DSML useful for complex real-world applications. Dynamic semantics of a DSML can be enforced using *interpreters*. An interpreter is a component that is written

using a traditional programming language like C++, Python or Java. Interpreters need to be registered with GME. When an interpreter is invoked, it is given access to the model hierarchy by GME, and can be used to perform different kinds of validation and generative operations on the models. One such operation can include generating platform-specific artifacts like code directly from the models.

III. Applying MIC to Develop Component-based Applications

This section describes a DSML called Platform-Independent Component Modeling Language (PICML). First we describe the challenges in system composition and integration when using component-middleware technologies without adequate tool support. Then we describe how we have applied the MDD approach to develop PICML, and show how the features in PICML help resolve the challenges with development, configuration and deployment of component-based applications.

A. Component-based Application Development Challenges

A common trend in the transition to component middleware technologies is the use of metadata to capture properties of applications that were previously tightly coupled with the implementation. This allows declarative specification of properties of applications using platform-agnostic technologies like XML, which are read by tools during deployment, and allows for automating the deployment and configuration of applications.

The availability of higher-level abstractions like virtual machines, execution containers, and extra information about systems via rich metadata, has an indirect effect in allowing people to build systems that are more heterogeneous than previously possible. This results in an increase in the amount of information that must be managed by the system developer as well as an increase in the complexity of system integration. However, the infrastructure for managing such complex deployment was essentially lacking in previous generation middleware. Most deployments were done in an *ad hoc* basis and there was hardly any reuse of the deployment infrastructure. Moreover, these *ad hoc* techniques had no basis for scientifically verifying and validating the correctness of the system. The amount of heterogeneity in the systems being deployed was also less compared to systems built using component-based middleware. The lack of simplification and automation in resolving the challenges outlined above can significantly hinder the effective transition to – and adoption of – component middleware technology to develop systems, thereby negating the benefits of component middleware technologies.

B. Platform-Independent Component Modeling Language

To address the problems outlined above, we have developed a DSML called Platform-Independent Component Modeling Language (PICML). PICML is an open-source DSML available for download as part of the CoSMIC MDD framework at `www.dre.vanderbilt.edu/CoSMIC`. PICML enables developers of component-based systems to define application interfaces, QoS parameters, and system software building rules, as well as generate metadata, 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 the interactions of their subsystems, 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 Deployment and Configuration Engine (DAnCE) [6], a QoS-enabled deployment engine. However, PICML's design has been driven by the goal to allow integration of systems built using different component technologies like Microsoft .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, ∼20,000 lines of C++ code (which represents the modeling language elements as equivalent C++ types) is generated. This generated code allows manipulation of modeling elements, *i.e.*, instances of the language types using C++, and forms the basis for writing *model interpreters*, which traverse the model hierarchy to generate XML-based deployment plan descriptors (described in Sidebar 2) needed to support the OMG Deployment and Configuraion (D&C) specification [7].

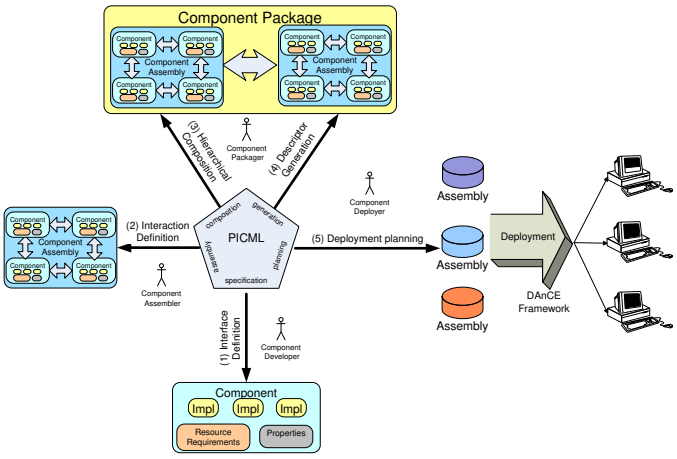

Fig. 3: Model-driven Application Development Lifecycle

Figure 3 shows the typical steps involved in developing component-based applications using PICML's MDD approach. The following describes the key steps in component-based application development, while describing the features in PICML used in this process:

1. Visual component interface definition. A set of component, interface and other datatype definitions defined via CORBA's Interface Definition Language (IDL) may be created in PICML using either of the following two approaches: (1) *Adding to existing definitions imported from IDL.* In this approach, existing CORBA software systems can be easily migrated to PICML using its *IDL Importer*, which takes any number of CORBA 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; (2) *Creating IDL definitions from scratch.* In this approach, PICML's graphical modeling environment provides support for designing the interfaces using an intuitive "drag and drop" technique making this process largely self-explanatory and independent of platform-specific technical knowledge. CORBA IDL can be generated from PICML enabling generation of software artifacts in languages having a CORBA IDL mapping.

2. Valid component interaction definition. By elevating the level of abstraction via MDD techniques, the well-formedness rules of DSMLs like PICML actually capture semantic information, such as constraints on composition of models, and constraints on allowed interactions. 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 programming language compiler. In PICML, OCL constraints are used to define the static semantics of the modeling language, thereby disallowing invalid systems to be built using PICML, *i.e.*, 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, and the limitations of human cognition. Without some form of support for hierarchical composition, observing and understanding system representations in a visual medium does not scale. To increase scalability, PICML defines a *hierarchy* construct, which enables the abstraction of certain details of a system into a hierarchical organization, such that developers can view their system at multiple levels of detail depending upon their needs. The support for hierarchical composition in PICML not only allows system developers to visualize their systems, but also allows them to compose systems from a set of 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. PICML therefore enables the development of repositories of predefined components and subsystems. The hierarchical composition capabilities provided by PICML are only a *logical* abstraction, *i.e.*, deployment plans generated from PICML flatten out the hierarchy to connect the two destination ports directly (which if not done will introduce additional overhead in the communication paths between the two connected ports), thereby ensuring that at runtime there is no extra overhead that can be attributed to this abstraction.

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 that are needed as input to the component deployment mechanisms. The descriptors generated by PICML conform to the descriptors defined by the standard OMG D&C specification [7]. Sidebar 2 shows an example of the types of descriptors that are generated by PICML, with a brief explanation of the purpose of each type of descriptor.

5. Deployment planning. Systems are often deployed in heterogeneous execution environments. To support these needs, PICML can be used to specify the target environment where the system will be deployed, which includes nodes, interconnects among nodes, and bridges among interconnects, all of which collectively represent the target environment. Once the target environment is specified via PICML, allocation of component instances onto nodes of the target target environment can be performed. PICML currently provides facilities for specifying static allocation of components.

Sidebar 2: Deployment Metadata

PICML generates the following types of deployment descriptors based on the OMG D&C specification:

- **Component Interface Descriptor (.ccd)** – Describes the interfaces – ports, attributes of a single component.
- **Implementation Artifact Descriptor (.iad)** – Describes the implementation artifacts (*e.g.*, DLLs and executables) of a single component.
- **Component Implementation Descriptor (.cid)** – Describes a specific implementation of a component interface; also contains component interconnection information.
- **Component Package Descriptor (.cpd)** – Describes multiple alternative implementations (*e.g.*, for different OSes) of a single component.
- **Package Configuration Descriptor (.pcd)** – Describes a component package configured for a particular requirement.
- **Component Deployment Plan (.cdp)** – Plan which guides the runtime deployment.
- **Component Domain Descriptor (.cdd)** – Describes the deployment target *i.e.*, nodes, networks on which the components are to be deployed.

IV. Applying MIC to develop embedded automotive applications

Embedded automotive systems are becoming notoriously difficult to design and develop. Over the past years there has been an explosion in the scale and complexity of these systems, owing to a push towards drive-by-wire technologies, increasing feature levels, and increasing capabilities in the embedded computing platforms. In order to address this level of complexity, the automotive industry has in general embraced the model-based approach for embedded systems development. However, the approach is confined to only the functional aspects of the system design, and restricted to a limited suite of tools, most notably the Mathworks family of Matlab®, Simulink® (SL), Stateflow® (SF) tools. Undeniably, Simulink and Stateflow are very powerful, graphical system design tools for modeling and simulating continuous and discrete event-based behavior of a dynamic system. However, these tools by no means cover the entire spectrum of embedded systems development. There are several other complex activities such as requirements specification, verification, mapping on to a distributed platform, scheduling, performance analysis, and synthesis in the embedded systems development process.

Although there are tools which individually support one or more of these other developmental activities, the integration among these tools and the Mathworks family of tools is

often lacking, which makes it extremely difficult to maintain a consistent view of the system as the design progresses through the development process, and also requires significant manual effort in creating different representations of the same system. To address the deficiencies in the development process for distributed automotive embedded systems, we built the Embedded Control Systems Language (ECSL) that provides the ability to import existing SL/SF models into a GME environment, and supports: (1) The annotation of structural design, SW-component design, and behavior implementation to supply information needed by a code generator, (2) Creation of HW-topology design models, Electronic Control Unit (ECU)-design models, and firmware implementation design models, and (3) Creation of deployment models that capture component and communication mapping.

A. *Embedded System Development using ECSL*

Figure 4 depicts the conceptual view of activities in an automotive embedded systems development process, as supported by the ECSL tools. Each rounded 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.

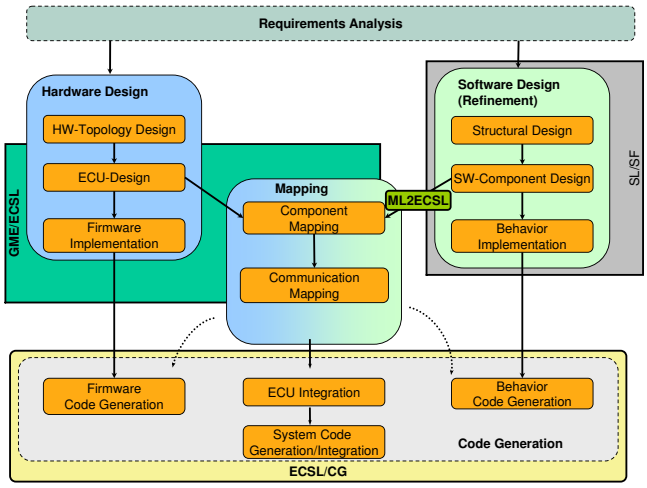

Fig. 4: ECSL Process

Software Design deals with: a) Structural design, which refers to the hierarchical decomposition of the embedded system into subsystems and sub-subsystems, from a functional viewpoint, b) Component design, which is another form of decomposition not independent of the functional decomposition, deals with more of the classical embedded software concerns such as real-time requirements, real-time tasks, periodicity, deadline, and scheduling, and c) Functional/behavioral design, which refers to the elaboration of the leaf elements of the hierarchical structural design in terms of a synthesizable realization.

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

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

Code Generation/Implementation involves creation of low-level coding artifacts, which include RTOS configuration, firmware configuration code, and behavioral implementation of the components.

Figure 4 captures an abstraction of the design process, which is made concrete by a number of supporting tools, which include: **GME/ECSL.** This is the Generic Modeling Environment tailored to support the ECSL modeling language. **ECSL/CG.** A specialized code generator that produces various production artifacts (*e.g.*, source code, configuration files) from ECSL models. **ML2ECSL.** Import translators that allow importing SL and SF models into the ECSL modeling environment. **SL/SF.** These are the Simulink and Stateflow tools.

B. *Embedded Control Systems Language*

ECSL is a graphical modeling language built using GME. It contains modeling concepts for specification of the design activities listed above. The language has been designed as a composition of a suite of sub-languages as shown below:

Functional Modeling. The Simulink and Stateflow sub-languages of ECSL were designed to mirror the capabilities found in SL and SF, such that all models could be imported into the GME environment configured to support ECSL. Simulink follows a dataflow-diagram like visual notation, while Stateflow supports Statechart-like hierarchical finite state machines.

Component Modeling. This sub-language allows software modeling in two stages: (1) integrating models that were imported from SL/SF, and (2) allowing their componentization. A component is defined as a portion of the software model, which is deployed as a unit. The 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). Components consist of ports which allow the specification of inter-component communication, as well as interfaces to physical devices including sensors and actuators. Attributes of ports capture the required communication properties like data type, scaling, and bit width. The intra-component dataflow exists within the functional models and it is imported from SL/SF. The inter-component dataflow is introduced by the designer after creating components from the imported SL/SF models.

Hardware Topology Modeling. The hardware modeling sub-language of ECSL allows the designer to specify the hardware topology, including the processors and communication links between the processors. ECU models represent specific processors in the system. An ECU model has two kinds of ports

for representing the I/O channels and the bus connections. I/O channel ports come in two variants: sensor ports and actuator ports. The specifics of the ECU firmware, characterization with respect to memory sizes and CPU speed, are captured with attributes. Bus models represent communication pathways used to connect ECUs. Buses are expressed as GME atoms and their attributes specify various properties of the physical communication system (*e.g.* bit rates).

Deployment (Mapping) Modeling. This modeling sub-language captures how software components are deployed on the hardware. The ECU model has a "deployment aspect" that allows the designer to capture SW component to ECU mapping using GME's reference concept. Note that deployment models are separate from software models, thus allowing the reuse of software models in different HW architectures. Furthermore, component ports are connected to ECU ports (sensor, actuators, and bus connections) to indicate how the component software interfaces map to actual sensors, actuators and buses.

C. Generating code and other artifacts

The ECSL/CG tool is a code generator that produces code artifacts necessary for system implementation as shown on Figure 5.

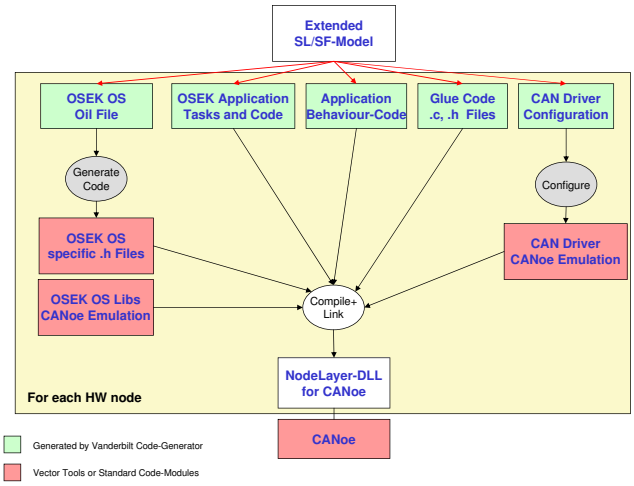

Fig. 5: ECSL Code Generation

The following types of files are generated: (1) **OSEK Implementation Language (OIL) File** - For each ECU-node in the network an OIL file is generated, that includes a listing of all used OSEK [8] objects and their relations, (2) **OSEK Tasks and Code** - All tasks are implemented in one or more C files, (3) **Application Behavior Code** - A separate function is generated for each application component that implements the behavior of the component. This function is called out from within a task frame, (4) **Glue Code** - The glue code comprises one or more C code/header files that resolve the calls to the Controller Area Network (CAN) driver or the firmware in order to provide access to CAN signals or HW I/O signals.

V. Related Work

This section summarizes related efforts associated with developing design environments supporting the MDD approach and compares these efforts with our work.

A. Cadena

Cadena [9] is an integrated environment developed at Kansas State University (KSU) for building and modeling component-based systems, with the goal of applying static analysis, model-checking, and lightweight formal methods to enhance these systems. Cadena also provides a component assembly framework for visualizing and developing components and their connections. Unlike PICML, however, Cadena does not support activities such as component packaging and generating deployment descriptors, component deployment planning, and hierarchical modeling of component assemblies. To develop a complete MDD environment that seamlessly integrates component development and model checking capabilities, we are working with KSU to integrate PICML with Cadena's model checking tools, so we can accelerate the development and verification of DRE systems.

B. Kennedy Carter Modeling Tool-suite

Kennedy Carter [10] provides a solution called iUML, which uses the MDA approach to develop systems using executable UML. iUML provides a full fledged modeling framework as well as a model testing, debugging and execution environment. iUML also provides a ready to use code generation framework so that developers can translate the models using a proprietary code generation framework. Like UML, iUML is also a generic framework and the developers will have to customize the tool to create domain-specific artifacts. It is possible that Kennedy Carter will target DSMLs for vertical domains which use this generic framework.

C. Eclipse Modeling Framework

Eclipse Modeling Framework(EMF) [11] is a modeling framework targeting the MDA approach to MDD. EMF uses MOF [12] as the underlying meta-metamodel. EMF supports specification of the models using XML Metadata Interchange (XMI), annotated Java, and XML Schema. EMF generates Java code that allows a user to manipulate the elements in a model. There are efforts like Graphical Modeling Framework (GMF) which are currently underway with the goal of adding capabilities for visual specification of DSMLs just like GME.

D. Microsoft DSL Tools

Microsoft has recently released a set of tools with the goal of realizing the vision of *Software Factories* [13]. Like PICML, the release includes a DSML for building .NET based Web Services applications called Visual Studio for Team Architects (VSTA), which provides tight integration with code, and generates descriptors necessary for deploying these applications. Microsoft is also developing another set of tools called the domain-specific language (DSL) tools. The DSL tools are like GME and EMF and allow development of DSLs using a proprietary meta-metamodel.

VI. Concluding Remarks

Model-driven development (MDD) is a promising paradigm to tackle the system composition and integration challenges. MDD elevates the level of abstraction of software development and bridges the gap between technology domains by allowing domain experts (who may not be experts in software development) to be able to design and build systems. The higher level of abstraction provided by MDD also allows transformation between models in different domains without the need to resort to low-level integration solutions like using standard network protocols. MDD guarantees the semantic consistency of the systems that are built by enforcing the philosophy of "correct-by-construction". MDD also solves a lot of the accidental complexities, due to the heterogeneity of the underlying component middleware technologies, that arise during system integration. For example, applying MDD to develop component-based applications using PICML relieves the users from having to capture the metadata in the form of XML descriptors, which are tedious and error-prone to write manually.

With improvements in generative techniques, MDD has the potential to automate the generation of code just like compilers replaced assembly language/machine-code programming. This allows people to augment their existing software methodologies with MDD, and helps in easing the transition from a pure coding based approach to a pure MDD approach.

References

[1] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, no. 4, pp. 110–112, Apr. 1997.

[2] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[3] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Indianapolis, IN: John Wiley and Sons, 2003.

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

[5] *Unified Modeling Language: OCL version 2.0 Final Adopted Specification*, OMG Document ptc/03-10-14 ed., Object Management Group, Oct. 2003.

[6] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, "DAnCE: A QoS-enabled Component Deployment and Configuration Engine," in *Proceedings of the 3rd Working Conference on Component Deployment*, Grenoble, France, Nov. 2005.

[7] *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 ed., Object Management Group, July 2003.

[8] OSEK, "Open systems and the corresponding interfaces for automotive electronics," www.osek-vdx.org/.

[9] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[10] K. Carter, "Kennedy Carter iUML 2.2," www.kc.com, 2004.

[11] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework*. Reading, MA: Addison-Wesley, 2003.

[12] *MetaObject Facility (MOF) 2.0 Core Specification*, OMG Document ptc/03-10-04 ed., Object Management Group, Oct. 2003.

[13] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. New York: John Wiley & Sons, 2004.