Applying MDA Modeling to Development of Real-Time Software

Using a model-driven architecture approach to developing real-time systems offers developers enhanced communication of the requirements from domain experts as well as more efficient development and the ability to re-use and modify existing code.

By Kelvin Nilsen and Michael Benkel, Aonix

The Unified Modeling Language (UML) provides standard abstractions to simplify the documentation, understanding and maintenance of complex software systems. Extensions to standard UML can be used to represent schedulability, performance and timing of real-time software. These extensions are described in UML Profile for Schedulability, Performance, and Time Specification (SPT), published by the Object Management Group (OMG) in 2002.

UML is a modeling language and not a programming language. As a modeling language, UML makes it possible for developers to communicate with system engineers who are experts in a specific domain in which the real-time computer system is embedded and who do not necessarily understand software engineering.

Standard UML is extended to project-specific needs (i.e., embedded systems) by the use of profiles. New semantics can be added to existing UML elements by defining stereotypes, tagged values, and constraints.

UML models quickly orient developers to the architecture (high-level structure and relationships) of a particular embedded system. For UML to serve effectively as a modeling tool, it is important that the UML diagrams present a high level of abstraction, omitting many low-level details of actual code. It is also important that they remain consistent with actual software implementations.

Far too often, architecture and design documents become invalid by architecture and design changes made to overcome difficulties and opportunities encountered when the system is coded and tested.

Over the years, UML users have found that the best way to achieve their goals is to adopt model-driven architecture (MDA) practices, in which UML models are automatically transformed from higher to lower levels of abstraction and finally into source code for the programming language of choice (Figure 1).
Model-Driven Architecture

MDA begins with a computation-independent model (CIM), which represents physical aspects of the system in terms understood by the domain practitioner, who is often not a computer specialist. The CIM facilitates communication between the domain experts and the system architects and might include, for example, use-case descriptions (e.g., how users expect to interact with the system) and sequence diagrams (e.g., sequences in which certain actions are performed).

After the CIM model is sufficiently developed, system architects build an object-oriented architecture to represent the planned solution. This architecture is represented by a platform-independent model (PIM) where resource requirements are given in abstract terms and represented by a variety of UML diagrams including use-case, sequence and class diagrams. Architects minimize dependencies on programming languages, RTOSs, etc.

The platform-independent model becomes the starting point for model-driven verification (MDV), which verifies consistency throughout the model. Through verification, design errors are identified early in the development cycle before teams proceed to implement erroneous designs. The system architect inserts assertions into the PIM, which the model checker then verifies. With traditional non-real-time MDA development environments, model checking verifies consistent type and visibility declarations for the operations and state variables represented in PIM drawings.

A key strength of the model-driven architecture is its ability to automatically transform the platform-independent model into a platform-specific model (PSM). System designers define transformation rules, which automatically translate the model into source code. Since UML modeling gives a high-level abstract representation of the complete software system, the generated code outlines a skeleton of the software system to be implemented. A state diagram, which describes the lifecycle of a class, contains enough implementation details to generate complete executable code. This level of detail automatically transforms into a complete executable implementation.

Ultimately, the system architect must decide how much detail to insert into the model. The MDA development approach is very powerful. By designing appropriate stereotypes and transformation templates, designers can automatically generate up to 100% of the source code for the architectural infrastructure. For the algorithm implementations, the generated code includes placeholders, which are subsequently replaced by the software engineers responsible
In UML notation, the platform-independent model is central to the MDA approach. Therefore, software engineers must practice self-discipline to maintain consistency between the UML model and their implementation of that model. In particular, whenever they insert code into the skeleton outline automatically produced from the UML model, they need to ensure that the generated code is consistent with the intent of the original placeholder they are replacing.

If there is no placeholder to represent the code they’re inserting, they need to request changes to the original UML model from the system architects and designers. (Notably, a single placeholder represents all of the source code relevant to the intent of the original placeholder and might consist of many lines of source code.) After the requested changes to the UML diagrams are incorporated, the code skeletons are automatically regenerated, preserving all of the placeholder replacements previously introduced by the software engineers. The newly generated skeletons include new placeholders that represent the new capabilities added to the system architecture. These concepts can be illustrated with two examples.

The first is the High Integrity Distributed Object-Oriented Real-Time System (HIDOORS) project developed by an industry consortium. In this example, the platform-specific model is represented as real-time Java source code. In the second example, the Object-Oriented Modeling concept for real-time Software (OMOS), developed by Bosch, is represented in C. In both projects, the same MDA models could be translated into alternative programming languages by replacing the translation templates.

HIDOORS

The HIDOORS project defines UML stereotypes and tagged values (e.g., vocabulary) to allow system architects to speak of periodic tasks, execution frequencies and expected worst-case execution times. The HIDOORS vocabulary also describes sporadic tasks, priority inheritance locks, priority ceiling locks and certain real-time message passing mechanisms.

When using MDA to support development of real-time systems, the HIDOORS architect can insert into the platform-independent model assertions about the:

- Number of real-time tasks
- Frequency of particular real-time task execution
- Anticipated worst-case execution times for each task
- Acceptable latency from asynchronous event stimulus to completed response

Along with the standard model consistency checking typical in traditional UML environments, the HIDOORS model checker also performs rate monotonic scheduling analysis to assure that all real-time tasks will satisfy their real-time constraints.

After the platform-independent model is transformed into the platform-specific model, the HIDOORS model checker also checks worst-case execution time analysis on the source code using values from the intended execution platform. The checker verifies that the actual execution times of tasks are consistent with the worst-case execution time budgets defined in the platform-independent model (Figure 2).
OMOS

The OMOS modeling environment is tailored to the development of cost-sensitive control systems software where demands on memory and CPU throughput require extreme efficiency. The automobile industry has been drawn to object-oriented modeling with its control software because it can efficiently support many different deployment variants. As you’ll see, by using object-oriented abstraction hierarchies, the same software framework can be custom-configured for either front- or rear-wheel drive, and can include optional support for anti-lock braking and traction control.

OMOS confines itself to a subset of UML-based object-oriented modeling, making it possible to automatically transform the UML to efficient C source code. Important object-oriented concepts like object instantiation, class definitions, inheritance and virtual methods are emulated in the generated C code. More costly object-oriented features, such as multiple inheritance and templates, are intentionally forbidden within the OMOS environment.
In OMOS, UML stereotypes for class symbols differentiate between 1-Class and N-Class. An N-Class is equivalent to the class perception known from object orientation and can be instantiated as often as desired. In contrast, there can be only one instance of a class defined with the 1-Class stereotype.

This characteristic provides great potential for optimization of a C implementation. All attributes of a 1-Class can be implemented as global variables, removing the need for de-referencing objects and copying object references. In typical automotive electronics systems, there are large numbers of 1-Classes that represent the motor, the gas peddle, the gearbox and so forth. Thus, this optimization yields significant benefits.

Figure 3 illustrates a simple control system model in the OMOS profile. The CL_VehicleData and CL_BrakePedal are both given the stereotype 1-Class, since only one instance of each is required for each vehicle. Axles and wheels, which involve multiple instances per vehicle, are both given the stereotype N_Class. In this drawing, the composition relationships labeled with the names FA, RA and WL, and WR specify that each vehicle is composed of exactly two instances of CL_Axle with two instances of CL_Wheel associated with each of those.

A variant is expressed in UML notation by an inheritance relationship. A new (class) variation is modeled as a sub-class of a more general class variation and it inherits the structure, behavior and interface of the general variation. It can be supplemented with additional attributes, methods and relationships and can overload the implementation of inherited methods. Inherited characteristics cannot be eliminated, as this would compromise conformance with the interface of the super class. The UML variant notations allow designers to describe pools of possible variations of control system software.
Figure 4 represents a generalization of the previous model, in which the axle and wheels are enhanced with anti-slip (traction control) capabilities. The illustration describes a relationship in which CL_AxleASR inherits from class CL_Axle, and shows that control of the anti-slip capabilities is implemented in the CL_ASR class. The model described by Figures 3 and 4 together provides the flexibility to build one car with automatic traction control and anti-lock braking, and a different car with just anti-lock braking.

Since OMOS models contain inheritance relationships that describe multiple possible configurations of the control software, it is necessary to select between the many available variations for each key component. Figure 5 offers a special UML diagram to represent the configuration that has traction control on the front axle. Instances of the classes CL_AxleASR and CL_ASR are configured for this purpose. The communication relationships ASR_WL, ASR_WR and the attribute maxSlip are all initialized in the case of the instance CL_ASR.
Model Transformation to Implementation Code

Because the transformation of UML models into implementation code in the OMOS environment is based on OMG’s model-driven architecture, all classes are generated consistently and the models are independent from the target environment. A future technology can transfer the model to embedded C++ or Java at a later time.

Special templates describe the mapping rules from UML models to C depending on the stereotype of a particular class. Code Listing 1 offers two examples that show portions of the implementation of the 1-Class CL_BrakePedal and the N-Class CL_Wheel introduced in Figure 3.

The example shows which classes will be instantiated based on the model and the selected configuration. Here, we have four wheels on two axles with the front axle implementing the ASR (traction-control) functionality.

OMOS and HIDOORS represent two implementations of modeling real-time software for embedded systems using UML and model-driven architecture. These approaches demonstrate that it is possible to automatically generate either C or Java code from the high-level architectural models. Model checking capabilities allow design inconsistencies to be identified and corrected early in the development process.

Several systems developed with the UML-MDA approach are in production now. Not surprisingly, development teams have noted the improved communication that UML has brought with the project team and off-project participants. Experience with OMOS demonstrates that MDA modeling allows developers to release new variants of existing vehicle control systems in approximately one third the time previously required.

Aonix
San Diego, CA.
(858) 457-2700.
[www.aonix.com].

© 2005 RTC Group, Inc., 905 Calle Amanecer, Suite 250, San Clemente, CA 92673