Rules-based Code Generation

The Right tool for the Job

Stephen Mellor lists three general ways a UML tool can be used in the software development process: sketching, blueprinting, and executing [1].

We define sketching to mean a development method in which part of the system is drawn using UML notation in hard or soft copy form as a means of organizing and communicating ideas. This is done as a preliminary step to writing code in the target development language. Once code development starts, the models are typically left behind as a result of the developer not wanting to keep the models completely up to date with the current state of the code. Sketching is the most common way UML is currently being used in the software development arena.

We define blueprinting to contain the sketching process, while adding to the drawn models details of the implementation using the target language. Modeling tools in this category go above and beyond sketching tools by providing the ability to export some target code skeletons for the modeled data. The tools also directly export the target implementation code the user added as processing in the model. Developers using these tools are more inclined to keep the models in synch with the implementation, but it is not required by the tool. Developers may also prefer to take the exported code and manage the rest of the development using an IDE they are more familiar and comfortable with.

A tool in the last category has the sketching and blueprinting capabilities, and abstracts the software under development away from the target code implementation details. It does this by specifying the processing in the application using a platform-independent language (PIL). When the developer is ready to run the software on the target, they use a model compiler and code generator to transform the data and control models, as well as the PIL-based processing into the target language.

Each of these three methods is a good step to take to add UML-based design and abstraction to the software development process. The remainder of this paper will focus on the special benefits the executable and translatable method can provide.

Benefits Explained

There are many benefits to raising the level of abstraction of software development. In much the same way that C compilers raised the level of abstraction away from assembly code, model compilers perform an analogous action for C or other implementation-oriented languages such as C++, Java, etc…

Coding Standards Enforcement
Coding standards have been around as long as software development teams have existed. Having clear and consistent code is invaluable to the maintainers in the long term.

Coding standards also aid the code review process, by allowing the reviews to focus on logic, not coding style. When all of the target code is generated, the team is instantly guaranteed a consistent style across the entire code base. This benefit is only realized by using a platform independent action language in the model. The team could also extend the code generation rules in customized ways, such as outputting a code generation error for all classes that do not have description documentation.

Abstracting Away Implementation Details

C compilers allowed assembly programmers to stop worrying about implementation details such as registers, branching and looping control, and the state of the stack; programmers across the world rejoiced. By relying on a PIL-based model compiler, programmers are freed even more of the implementation details and allowed to focus their attention on the design at hand. Does a hash or list make sense for this data? Should these instances be kept in an ordered list? What threading is necessary and/or appropriate for this functional block? These types of details can be tailored in the code generation rules, but are no longer of equal concern to the details of the functionality to be implemented. In some cases, teams can be split into application specialists and implementation specialists. The application specialists are completely free to specify the functionality of the system using their domain-specific knowledge. The implementation specialists can focus, as necessary, on tailoring the code generation rules to meet the tradeoff demands that go hand-in-hand with embedded control applications.

Customization

Because each model construct has a mapping to a target language construct, the code generation process is 100% repeatable. All modeled elements are guaranteed to have identical implementation layouts. This means that the developer can implement an optimization in the code generation rules and immediately see the optimization applied throughout the entire code base. Or, perhaps a specific target constraint needs to be addressed. One change to the code generation rules will here also be recognized instantly across the entire code base.

A model compiler can take flags (also called marks) as input to control many features: memory management, debugging and tracing, threading type and multiplicity, etc. Again, these marks are applied instantly and easily to the entire generated code base when the model compiler is next run.

Architecture Reuse

All model compilers generate a certain amount of software architecture along with the modeled application. Because this architecture is an inherent part of the generated code, it is easily reused across many projects. When choosing a modeling tool, developers should make sure the architecture is designed not to get in their way if the team chooses to integrate with any of the popular hard and soft real-time operating systems available today.

Standards Body Compliance

In more and more industries, compliance with the rules and guidelines of standards bodies such as MISRA and IEC65801 is increasing critical. This is one area where the PIL-based model compilers significantly outpace normal blueprinting UML tools.

We've already discussed applying optimizations or target constraints to the code generation rules and how the impact on the generated code is immediate. The same is true for standards body rules. The code generation rules can be modified to implement a standards rule in *one-place*, and be instantly applied to the entire generated code base. Each developer on the team does not then have to be an expert on the compliance rules, during development or code review. This can save the team countless man-hours in training, development, and review. A blueprinting tool provides this benefit only on the portion of structural code the tool generates. Since it puts the target code inside the processing model, it cannot provide this benefit universally to the entire application or to the development team. Only by relying on platform-independent action language in the processing model and implementing the compliance rules in the code generation rules will this benefit be fully realized.

References

[1] Mellor, Stephen J. "Demystifying UML", Embedded Systems Design, March 2006: http://www.embedded.com//showArticle.jhtml?articleID=180205522