The Pragmatics of Model-Driven Development

Bran Selic, *IBM Rational Software*

Using models to design complex systems is de rigeur in traditional engineering disciplines. No one would imagine constructing an edifice as complex as a bridge or an automobile without first constructing a variety of specialized system models. Models help us understand a complex problem and its potential solutions through abstraction. Therefore, it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modeling techniques. However, for historical reasons, models in software engineering are infrequent and, even when used, they often play a secondary role. Yet, as we shall see, the potential benefits of using models are significantly greater in software than in any other engineering discipline.

Model-driven development methods were devised to take advantage of this opportunity, and the accompanying technologies have matured to the point where they are generally useful. A key characteristic of these methods is their fundamental reliance on automation and the benefits that it brings. However, as with all new technologies, MDD's success relies on carefully introducing it into the existing technological and social mix. To that end, I cite several pragmatic criteria—all drawn from industrial experience with MDD.

Model-driven development holds promise of being the first true generational leap in software development since the introduction of the compiler. The key lies in resolving pragmatic issues related to the artifacts and culture of previous generations of software technologies.

The challenge

Software engineering is in the unfortunate position of being a new and relatively immature branch of engineering of which much is expected. Seduced by the relative ease of writing code—there is no metal to bend or heavy material to move—and compelled by relentless market pressures, software users and developers are demanding systems whose complexities often exceed our abilities to construct them.

This situation is not without precedent in the history of technology; similar situations occurred when the Industrial Revolution introduced new technologies such as steam and electrical power.[1] What seems to be unique,

0740-7459/03/$17.00 © 2003 IEEE

Many practitioners have given up all hope that significant progress will result from fundamental advances in programming technologies.

however, is how slowly software technologies have evolved to meet the obvious need for improving product reliability and productivity.

In particular, since the introduction of third-generation languages in the late 1950s, the essence of programming technology has hardly changed. Although we've introduced several new programming paradigms since then—such as structured and object-oriented programming—and much work has been done to polish the details, the level of abstraction of market-dominant programming languages has remained almost constant. An If or Loop statement in a modern programming language such as Java or C++ is not that much more potent than an If or Loop statement in early Fortran. Even promising mechanisms such as classes and inheritance, which have potential for producing higher forms of abstraction, remain underused. Objects, for example, are relegated to relatively fine-grained abstractions confined to a single address space (such as stacks, data structures, or graphic primitives) consistent with the granularity and abstraction level of the languages in which they appear.

In an industry that prides itself on its rapid advances, this apparent reluctance to move forward despite an obvious need might seem surprising. However, consider the sheer scale of investment—fiscal and intellectual—in those early-generation technologies. There are countless lines of code written in traditional programming languages that programmers must maintain and upgrade. This, in turn, creates a continuous demand for professionals who are trained in and culturally attuned to these technologies. Because of their intricate nature, attaining competency in such programming technologies requires significant investments in time and effort. This, quite understandably, fosters a conservative mindset in both individuals and corporations. Unless we properly account for such factors, no technical breakthrough is likely to succeed, regardless of how advanced and promising it might be.

Many practitioners have given up all hope that significant progress will result from fundamental advances in programming technologies; instead, they are placing their hopes on process improvements. This partly explains the current surge of interest in methods such as Extreme Programming and the Rational Unified Process.[2]

Although following a proper process is critical to any engineering endeavor's success, it's too soon to discount the possibilities that new programming technologies can achieve. After all, software development consists primarily of expressing ideas, which means that our ability to devise suitable facilities is mostly limited by our imagination rather than by unyielding physical laws. Taking advantage of this opportunity is one of the central ideas behind MDD and one of the reasons why it represents the first true generational shift in basic programming technology since the introduction of compilers.

I recognize that similar software "revolutions" have been proclaimed many times in the past but have had little or no fundamental impact in the end. Is there any reason to expect otherwise in this case? After all, MDD is based on the old idea of modeling software—a technique that has produced more than its share of skeptics.

The essentials

MDD's defining characteristic is that software development's primary focus and products are models rather than computer programs. The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages. This makes the models easier to specify, understand, and maintain; in some cases, it might even be possible for domain experts rather than computing technology specialists to produce systems. It also makes models less sensitive to the chosen computing technology and to evolutionary changes to that technology (the concept of *platform-independent* models is often closely connected to MDD).

Of course, if models end up merely as documentation, they are of limited value, because documentation all too easily diverges from reality. Consequently, a key premise behind MDD is that programs are automatically generated from their corresponding models.

As noted, however, both software modeling and automatic code generation have been tried before, meeting with limited success at best and mostly in highly specialized domains. But things have progressed since the early days. Aside from the fact that we now better understand how to model software, MDD is more useful today because of two key evolutionary developments: the necessary automation tech-

nologies have matured and industry-wide standards have emerged.

Automation technologies

Automation is by far the most effective technological means for boosting productivity and reliability. However, most earlier attempts at applying automation to software modeling were limited to "power-assist" roles, such as diagramming support and skeletal code generation. These are often not substantive enough to make a significant difference to productivity. For example, once the code is generated, the models are abandoned because, like all software documentation, they require scarce and expensive resources to maintain. This is why solutions based on so-called *round-trip engineering*, which automatically converts code back into model form, are much more useful. One drawback here, though, is that an automated conversion from code to model usually can't perform the kind of abstraction that a human can. Therefore, we can attain MDD's full benefits only when we fully exploit its potential for automation. This includes

- Automatically generating *complete* programs from models (as opposed to just code skeletons and fragments)
- Automatically verifying models on a computer (for example, by executing them)

Complete code generation simply means that modeling languages take on the role of implementation languages, analogous to the way that third-generation programming languages displaced assembly languages. With complete code generation, there is rarely, if ever, a need to examine or modify the generated program directly—just as there is rarely a need to examine or modify the machine code that a compiler produces.

Automatically verifying models means using a computer to analyze the model for the presence of desirable properties and the absence of undesirable ones. This can take many different forms, including formal (mathematical) analyses such as performance analysis based on queuing theory or safety-and-liveness property checking. Most often, though, it means executing (simulating) models on a computer as an empirical approach to verification. In all cases, it is critical to be able to do this on highly abstract and incomplete models that arise early in the development cycle, because this is when software designers make most of the fundamental design decisions.

The techniques and tools for doing this successfully have now reached a degree of maturity where this is practical even in large-scale industrial applications. Modern code generators and related technologies can produce code whose efficiency is comparable to (and sometimes better than) hand-crafted code. Even more importantly, we can seamlessly integrate such code generators into existing software production environments and processes. This is critical because it minimizes the disruption that occurs when MDD is deployed.

Standards

The last decade has seen the emergence of widely supported industry standards, such as those that the Object Management Group provides. The OMG is a consortium of software vendors and users from industry, government, and academia. It recently announced its Model-Driven Architecture initiative, which offers a conceptual framework for defining a set of standards in support of MDD (see www.omg.org/mda/index.htm). A key MDA standard is the Unified Modeling Language, along with several other technologies related to modeling.[3–5] In addition, other formal and de facto standards, such as various Web standards (XML, SOAP, and so forth) are also major enablers of MDD.

Standardization provides a significant impetus for further progress because it codifies best practices, enables and encourages reuse, and facilitates interworking between complementary tools. It also encourages specialization, which leads to more sophisticated and more potent tools.

Still, with all the benefits of automation and standardization, model-driven methods are only as good as the models they help us construct.

Despite all the benefits of automation and standardization, model-driven methods are only as good as the models they help us construct.

The quality of models

Models and modeling have been an essential part of engineering from antiquity (Vitruvius, a Roman engineer from the first century B.C., discusses the effectiveness of models in the world's oldest known engineering textbook[6]). Engineering models aim to reduce risk by helping us better understand both a complex problem and its potential solutions before undertaking the expense and effort of a full implementation. In