

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.

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,

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.

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 criti-

cal 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.

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 textbook⁶). 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

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

The rejection of modeling for software is ironic when you consider that software is the engineering medium best positioned to benefit from it.

contrast, large software projects typically involve great uncertainty about the design's viability until the final implementation phases—unfortunately, this is when the cost of fixing fundamental design flaws is greatest.

To be useful and effective, an engineering model must possess, to a sufficient degree, the following five key characteristics. The most important is *abstraction*. A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for a given viewpoint, it lets us understand the essence more easily. Considering the steady demand for ever-more sophisticated functionality from our software systems, abstraction is almost the only available means of coping with the resulting complexity.

The second key characteristic is *understandability*. It isn't sufficient just to abstract away detail; we must also present what remains in a form (for example, a notation) that most directly appeals to our intuition. Understandability is a direct function of the expressiveness of the modeling form used (expressiveness is the capacity to convey a complex idea with little direct information). A good model provides a shortcut by reducing the amount of intellectual effort required for understanding. One reason why programs are not particularly expressive, even when based on languages that support sophisticated abstractions, is that they require too much detailed parsing of text to be properly understood. Classical programming statements assault the reader with a profusion of syntactical detail assembled according to intricate lexical rules. The amount of information that must be absorbed and recognized to understand linear programs is enormous and requires significant intellectual effort.

The third key characteristic of useful models is *accuracy*. A model must provide a true-to-life representation of the modeled system's features of interest.

Fourth is *predictiveness*. You should be able to use a model to correctly predict the modeled system's interesting but nonobvious properties, either through experimentation (such as by executing a model on a computer) or through some type of formal analysis. Clearly, this depends greatly on the model's accuracy and modeling form. For instance, a mathematical model of a bridge is much better at predicting the maximum allowable load on a bridge than

even a very precise and detailed scale model constructed out of balsa wood.

Finally, a model must be *inexpensive*—that is, it must be significantly cheaper to construct and analyze than the modeled system.

Probably the main reason why software modeling techniques had limited success in the past is that the models often failed to meet one or more of the criteria just listed. In particular, the techniques tended to be weak in terms of accuracy (which also meant that the models weren't very useful for prediction). In part, this is because it wasn't always clear how the concepts used to express the models mapped to the underlying implementation technologies such as programming language constructs, operating system functions, and so forth. This semantic gap was exacerbated if the modeling language was not precisely defined, leaving room for misinterpretation.

Also, because the models weren't formally connected to the actual software, there was no way of ensuring that the programmers followed the design decisions captured in a model during implementation. They would often change design intent during implementation—thereby invalidating the model. Unfortunately, because the mapping between models and code is imprecise and the code is difficult to comprehend, such digressions would remain undetected and could easily lead to downstream integration and maintenance problems. (Changing design intent isn't necessarily a bad thing, but it is bad if the change goes unobserved.) Given these difficulties, many software practitioners felt that software models were untrustworthy, merely adding useless overhead to their already difficult task.

This rejection of modeling for software is particularly ironic when you consider that software is the engineering medium best positioned to benefit from it. This is because it is possible to gradually evolve an abstract software model into the final product through a process of incremental refinement, without requiring a change in skills, methods, concepts, or tools. The advantage of this is self-evident: there are no risk-laden semantic gaps to overcome when transferring a design into production. Model accuracy is guaranteed because the model eventually becomes the system that it was modeling. Furthermore, it is particularly conducive to an incremental iterative development style that is optimal when building complex engineering systems, because there

are no conceptual discontinuities that preclude backtracking. This unique property of software models is another cornerstone of MDD.

The pragmatics

Many software practitioners, when first faced with the notion of MDD, express concern about the technical difficulties involved in translating models into code. Will the code be fast enough and compact enough? Will it be a correct rendering of design intent? These, of course, are the very same questions that were asked when compilers were introduced more than 40 years ago. Although they were valid questions to ask at the time, it is worth noting that hardly anyone questions compiler technology these days because it is quite mature and extensively proven in practice.

In fact, experience with MDD in industrial settings indicates that code efficiency and correctness, although very important, are not the top-priority or even the most technically challenging issues associated with MDD. In fact, most standard techniques used in compiler construction can also be applied directly to model-based automatic code generation.

Model-level observability

Like all compilers, automatic code generators are idiosyncratic and often generate program code that, as a result of various internal optimizations, is not easily traceable to the original model. Thus, if an error is detected in the generated program, finding the place in the model that must be fixed either at compile time or runtime might be difficult. In traditional programming languages, we expect compilers to report errors in terms of the original source code and, for runtime errors, we now expect a similar capability from our debuggers.

The need for such facilities for models is even greater because the semantic gap between the modeling language's high-level abstractions and the implementation code is wider. This means that model-level error reporting and debugging facilities (in essence, "decompilers") must accompany practical automatic code generators. Otherwise, the practical difficulties encountered in diagnosing problems could be significant enough to nullify much of MDD's advantage. Programmers faced with fixing code that they don't understand will easily break it and will likely be discouraged from relying on models in the future.

This is a particularly important factor to consider for model-driven systems that are based on the notion of customizable transformation "templates." Such templates capture rules for translating models into corresponding code. By exposing these to developers, it is possible to streamline the generated code for specific target environments. This is a highly appealing and useful capability, but it must be matched by a similar facility for specifying inverse transformations, or model observability will most certainly be an issue.

Also related to model-level observability are two other critical facilities: model mergers and model difference tools. These tools are typically an integral part of configuration management systems that help us track different versions of the same model. Model merging tools merge two or more possibly overlapping models into one. In contrast to source-code merging used for traditional text-based programming languages, a model-level merge is much more complex because it requires a deeper understanding of the more complex semantics of the modeling language. The result must be a well-formed model. Furthermore, the tools must report any problems in a form that is meaningful to the modeler. Model difference tools identify the difference between two models (usually two different versions of the same model). They too must work at a semantically meaningful level.

Model executability

One of the fundamental ways that we learn is through experimentation—that is, through model execution (David Harel compares models that can't be executed to cars without engines). One important advantage of executable models is that they provide early direct experience with the system being designed. (When learning a new programming language, we are always inspired by the successful run of our first trivial "hello world" program. Simple as it is, that experience raises our confidence level and gives us a reference point for further exploration.) The intuition gained through experimentation is the difference between mere formal knowledge and understanding.

A common experimental scenario with executable models involves refining some high-risk aspect of a candidate solution down to a relatively fine level of detail, while other parts of the model remain sketchy or even undefined. This means that even incomplete models

Experience with MDD in industrial settings indicates that code efficiency and correctness are not the primary challenges of MDD.

It is now common knowledge that modern optimizing compilers can outperform most practitioners when it comes to code efficiency.

should be executable, as long as they are well formed. It also requires suitable runtime system support: the ability to start, stop, and resume a model run at any meaningful point; to “steer” it in the desired direction by simulating inputs at appropriate points in space and time; and to easily attach automated instrumentation packages to it. Finally, it is also extremely useful for developers to be able to execute a model in a simulation environment (for example, on a development workstation), on the actual target platform, or—and this is the most useful—on some combination of the two.

Efficiency of generated code

As mentioned earlier, one of the first questions asked about MDD is how the automatically generated code’s efficiency compares to handcrafted code. This is nothing new; the same question was asked when compilers were first introduced. The concern is the same as before: humans are creative and can often optimize their code through clever tricks in ways that machines cannot. Yet, it is now common knowledge that modern optimizing compilers can outperform most practitioners when it comes to code efficiency. Furthermore, they do it much more reliably (which is another benefit of automation).

We can decompose code efficiency into two separate areas: performance (throughput) and memory utilization. Current model-to-code generation technologies can generate code with both performance and memory efficiency factors that are, on average, within 5 to 15 percent (better or worse) of equivalent manually crafted systems. And, of course, we can only expect the situation to improve as the technology evolves. In other words, for the vast majority of applications, efficiency is not an issue.

Still, there might be occasional critical cases where manually crafted code might be necessary in specific parts of the model. Such hot spots are often used as an excuse to reject MDD altogether, even when it involves a very small portion of the complete system—the proverbial “baby and bathwater” scenario. A useful MDD system will allow for seamless and overhead-free embedding of such critical elements.

Scalability

MDD is intended for—and most beneficial in—large-scale industrial applications. This sometimes involves hundreds of developers working on hundreds of different but related

parts of a model, and the tools and methods must scale up to such situations.

The important metrics of concern here are *compilation time* and *system size*. We can divide compilation time into two separate parts: *full-system generation time* and the *turnaround time* for small incremental changes. Perhaps surprisingly, the latter is much more important because of its greater impact on productivity. Namely, small changes are far more frequent during development than full-system recompiles. Therefore, if a small, localized change requires regenerating a disproportionately large part of the code, development can slow to an unacceptable pace. This is particularly true in the latter phases of the development cycle, when programmers make many small changes as they fine-tune the system. To keep this overhead low, it is crucial for the code generators to have sophisticated change impact analysis capabilities that minimize the amount of code regeneration.

We can divide the system generation process into two phases. First, code generators translate the model into a program in some programming language and then compile the program using standard compilers for that language. After compiling the code, they link it to the appropriate libraries in the usual way. With modern automatic code generation technology, the compilation phase is significantly longer. Typically, compilation is an order of magnitude longer than code generation. This means that the overhead of automatic code generation is almost negligible compared to the usual overhead of compilation.

Regarding size, the largest systems developed to date using full MDD techniques have involved hundreds of developers working on models that translate into several million lines of standard programming language.


Integration with legacy environments and systems

A prudent and practical way to introduce new technology and techniques into an existing production environment is to apply them to a smaller-scale project such as a relatively low-profile extension to some legacy system. This implies not only that the new software must work within legacy software but also that the development process and development environment used to produce it must be integrated into the legacy process and legacy development environment.

This is not only a question of mitigating risk

but also of leveraging previous (usually significant) investments into such processes and environments. For example, a useful MDD tool should be able to exploit a range of different compilers, build utilities, debuggers, code analyzers, and software versioning control systems rather than requiring the purchase of new ones. Furthermore, this type of integration should work “out of the box” and should generally not require custom “glue” code and tool expertise. Fortunately, most legacy development tools have evolved along similar lines, supporting similar usage paradigms so that it is usually possible to construct MDD tools that can access these capabilities in a generic fashion.

Last but not least, an MDD project must be able to take advantage of legacy code libraries and other legacy software. These often capture domain-specific knowledge garnered over many years and often represent an organization’s prime intellectual property. This can be accomplished either using customizable code generators or by allowing direct calls to such utilities from within the model. For example, a model that uses Java to specify the details of actions along a statechart transition can simply make the appropriate Java calls without any intervening translation or having to go through a layer interface.

MDD’s success is not predicated only on resolving obvious technical issues such as defining suitable modeling languages and automatic code generation. Our experience with these methods in industrial environments on large-scale software projects clearly indicates that solving the unique pragmatic issues described in this article is at least equally, if not more, important.⁷ Unless the experience of applying MDD is acceptable from the day-to-day perspective of the individual practitioner and project manager, it will be rejected despite its obvious potential for yielding major productivity and reliability benefits. Fortunately, over the past decade, numerous commercial vendors have developed tools that address these issues successfully. The time for MDD has come. 

References

1. R. Pool, *Beyond Engineering: How Society Shapes Technology*, Oxford Univ. Press, 1997.

About the Author



Bran Selic is principal engineer at IBM Rational Software in Kanata, Ontario, Canada. He is also cochair of the OMG task force that is finalizing the UML 2.0 modeling language standard. He received his Magister Ing. Degree in systems theory and Dipl. Ing. Degree in electrical engineering from the University of Belgrade, Yugoslavia. He is a member of the IEEE and ACM. Contact him at IBM Rational Software, 770 Palladium Dr., Kanata, Ontario, Canada K2V 1C8; bselic@ca.ibm.com.

2. P. Kruchten, *The Rational Unified Process*, Addison-Wesley, 1999.
3. *Unified Modeling Language*, ver. 1.4, Object Management Group, 2002.
4. *Meta-Object Facility (MOF)*, ver. 1.4, Object Management Group, 2002; www.omg.org/cgi-bin/doc?formal/2002-04-03.
5. *Common Warehouse Metamodel (CWM) Specification*, ver. 1.1, Object Management Group, 2003; www.omg.org/cgi-bin/doc?formal/03-03-02.
6. Vitruvius, *The Ten Books on Architecture*, Dover Publications, 1960.
7. B. Selic, G. Gullekson, and P.W. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

New from The MIT Press

Software Ecosystem

Understanding an Indispensable Technology and Industry

David G. Messerschmitt and Clemens Szyperski

“Required reading for any serious student of the computer industry and its effects on business, innovation, and economic growth.”

— Nicholas Economides, New York University, and Director, NET Institute

432 pp., 49 illus. \$45



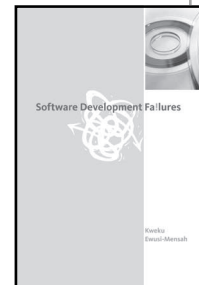
Software Development Failures

Kweku Ewusi-Mensah

“Makes a compelling argument for learning from software development failures, so that the same mistakes aren’t repeated in future projects.”

— Mark Keil, Georgia State University

288 pp., 5 illus. \$35



To order call **800-405-1619**.
Prices subject to change without notice.

<http://mitpress.mit.edu>