Model-Driven Development: Its Essence and Opportunities

Bran Selic
IBM Canada
bselic@ca.ibm.com

Abstract

This paper gives a short overview of model-driven development – an approach to software development in which high-level models play a fundamental role. First, the problems that plague current programming approaches are analyzed. This is followed by a short description of the essential features of model-driven development and the potential benefits that it can bring. The results achieved in industrial practice are reviewed briefly, followed by an analysis of current impediments towards greater levels of adoption of this approach and what needs to be done about to overcome them.

1. Introduction

When the AT&T long-distance network in the northeastern United States crashed in 1990, it paralyzed some of the world's key financial and business institutions. Consequently, the cost of this unfortunate incident was measured in the hundreds of millions of dollars. Yet, the root cause was traced to a single missing "break" statement in a C program – the kind of coding error that is quite common in practice and which is very difficult to detect, even with extensive testing and inspection.

We have come to accept that this is in the nature of software and that, on occasion, even the minutest of coding errors such as this one can have unexpected consequences. After all, these are extremely complex systems providing very complex functionality and comprising millions of lines of code. They are notoriously difficult to specify, let alone to implement correctly. Furthermore, the essence of formal logic – which is at the core of all programming – is binary: it is either correct or incorrect and a single flaw can invalidate an argument no matter how complex.

On the other hand, viewed from an engineering perspective, we must conclude that there is something seriously wrong with a technology in which it is so easy and so highly likely to make catastrophic errors of this kind – particularly a technology that is becoming more and more central to our everyday lives.

Without doubt, the key problem facing software designers and developers is complexity. Software systems are among the most complex systems ever constructed by man and the general trend, stemming from competitive market pressures and the incessant pursuit of greater productivity, is to make such systems even more complex. So, unless we do something about this unbridled complexity, our troubles with constructing software can only get worse.

In his excellent book on the experience with the development of the operating system for the IBM 360 series of computers, *The Mythical Man-Month*, Fred Brooks identified two kinds of complexity [1]. *Essential complexity*, is complexity that is inherent to and inseparable from the problem, such as the computational complexity of the traveling salesman problem. No matter what we do or how hard we try, this complexity cannot be eliminated. On the other hand, *accidental complexity*, is complexity that is a direct consequence of the way in which we choose to solve a problem. For example, trying to construct a skyscraper without any use of modern power tools such as bulldozers and hydraulic lifting devices, although theoretically feasible, is likely to be a very complex task. Clearly, our only possible solution to dealing with increasing complexity is to minimize the degree of accidental complexity. This is the central theme behind *model-driven development* (MDD) approach to software construction.

In the following section, we examine the various forms of accidental complexity that occur in traditional programming technologies and their impact on development. In section 3, we introduce MDD and reveal its essential aspects. We also explore the possibilities that lie behind this approach and how they can be exploited to yield software that is of much higher quality than is the current norm. Section 4 contains a brief review of results currently achieved in applying MDD to industrial software projects. Finally, in section 5, we discuss the issues facing the software industry as it moves towards more pervasive use of MDD.

2. The thesis

Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing
0-7695-2561-X/06 $20.00 © 2006 IEEE

The dominant programming languages of today, Java, C, C#, Visual Basic, etc., are all based on the conceptual foundations created for the earliest "high-level" programming languages in the late 1950's such as Fortran and Cobol. And, although there have been many developments since those early days, such as the introduction of the object paradigm, the functionality of a typical line of code written in a "modern" language such as Java is not significantly greater than the functionality of a line of Fortran. Yet, as noted, the complexity we are expecting of our software today often far exceeds what was sought when these languages were being devised. In essence, we are being asked to solve 21st century problems with mid-20th century technology.

A key thesis of this paper is that most of today's programming languages have an excess of accidental complexity. Specifically:

1. Programs written in these languages are extremely difficult to write and understand. Even programs comprising several hundred lines of code are a challenge in this regard, let alone programs containing millions of lines of code.
2. These programming languages are *defect intolerant*, in the sense that even the smallest defects can cause major and expensive failures. Unfortunately, as the network failure discussed above illustrates, these defects are both likely and difficult to detect.

2.1 Difficulties in understanding programs

Practically everyone who has written software using one of today's programming languages is familiar with how difficult it is to both write correct programs and read and understand them. This is because the complex real-world phenomena and activities that software typically deals with (e.g., telephone calls or insurance claim processing), are often very far removed from the statements and data structures that constitute the basic vocabulary of traditional programming languages. This semantic gap often needs to be bridged by numerous layers of abstraction. At the very lowest level is the computing platform, which is itself layered into hardware, operating system, and, possibly, some kind of middleware or programming framework. On top of this come the programming language or languages and their usage patterns. Finally, there will be one or more domain-specific layers in which the domain-specific concepts emerge through some composition of lower-layer concepts and capabilities.

Trying to author or understand a program written in this fashion requires knowledge of and experience with each these levels as well as the relationships between them. This can be a daunting intellectual challenge and does not come easily even to experts.

Consider, for instance, the realization of simple finite state machines using standard programming language constructs. These are typically implemented by sets of nested conditional statements. While this does not seem too technically challenging, the fact remains that it requires a transformation of the original intuitive graphical form into something less obvious and less intuitive. Furthermore, when such a transformation is manually programmed by humans, there is always the possibility of a basic coding error (such as accidentally omitting a "break" statement). This type of translation tends to be mostly mechanistic and is often tedious and slow. The situation is even worse when more powerful specification formalisms such as statecharts are used, since they typically have more complex semantics which are not easy to translate correctly.

2.2 Defect intolerance

The missing "break" problem described earlier that caused the telephone network to fail is just one example of the "chaotic" nature of today's programming language constructs. Even though attempts have been made to mitigate this (such as the infamous "go to"), it still remains too easy to make potentially catastrophic mistakes (e.g., accidentally using assignment in place of the logical equivalence operators, forgetting to initialize variables, forgetting to return memory after it is no longer needed leading to memory leaks, incorrect pointer assignments, concurrency conflicts, and so on).

As noted, the consequences of such seemingly minute errors can be enormous. Unfortunately, it is difficult to predict where they are likely to occur (only that they *are* likely) or to understand their effects with any degree of confidence or precision. From an engineering point of view, software written by human programmers represents a highly sensitive system in which every single detail is potentially significant because it may be pivotal to the correctness of the entire system.

2.3 Barriers to abstraction

This detailed nature of programming languages has another important drawback: it impedes abstraction. When systems get beyond a relatively simple degree of complexity, we turn to abstraction – the removal of "irrelevant" detail. This permits our attention to focus on the essential parts of a system. In fact, abstraction is really the only effective means that we have for coping with complexity. But, in an engineering system where every single detail is potentially significant, relying on

abstraction is dangerous, because one can never be sure that something fundamental has not been abstracted out.

One major detrimental effect of this is that it is often impractical to perform accurate analyses of software systems using that mainstay of all other engineering disciplines: formal mathematical methods. This is because most formal methods work with abstract (but sufficiently accurate) models and they do not scale up well when the models that are to be analyzed become too complex. Without the predictive power that mathematical analyses provide, software developers rarely know in advance if their designs will work until they are finally realized. This is, clearly, a very expensive and a very undesirable way of constructing systems.

2.3 What can be done?

The solution to reducing this accidental complexity is quite straightforward:

1. Use languages that have only well-behaved constructs;
2. Raise the level of abstraction of program specification to a level that is closer to the problem domain.

In the following section, we explain how MDD achieves these objectives.

3. Model-driven development

In general, the use of models in more traditional engineering disciplines has a long and highly successful history. Structural engineers, for example, use mathematical models to predict the load-bearing properties of their designs. Since the theory of the construction of such models is well established, the models generally tend to be accurate and the results of formal analyses on them are trustworthy.

Yet, there is a great deal of skepticism among software practitioners about the value of modeling software. There is some justification for this. As discussed above, models of software tend to be inaccurate because they are typically not amenable to abstraction. In addition, because models are translated into code by hand, there is no formal link between the realized software and its model. As a result, the two tend to drift apart. Trying to maintain the model to track the implementation involves overhead with seemingly no immediate value to the implementation team, who are typically working under pressure of tight delivery deadlines. The result is that models tend to be inaccurate and, therefore, like all documentation, not trustworthy.

Another problem with software models is that many modeling languages, particularly those of earlier generations, had very weak semantic specifications. They could be interpreted in a number of different and sometimes contradictory ways, which led yet again to divergence between design intent expressed in the model and its implementation in code.

These issues have led many to question the practical value of software modeling. Perhaps the dominant view today is that models may have value in the early phases of development to "sketch out" and communicate high-level design ideas [2]. Once that stage has been passed, conventional wisdom says that models only play a secondary, mostly documentational, role

Yet, as argued below, it seems that software is potentially the near-perfect medium for modeling. Unfortunately, this potential remains inadequately recognized by the greater software engineering community, and, hence, has not been sufficiently exploited. The good news is that the technologies required to fulfill this potential (particularly tooling) have been maturing over the past decade to the point where MDD is a viable alternative to traditional development in most application domains.

3.1 Engineering models

To understand what models can do for software development, we need first to understand how models are used in engineering.

An *engineering model* of some system is a reduced representation of that system that highlights the properties of interest for a given viewpoint. Clearly, a model is an abstraction – which is why it exists in the first place. A useful engineering model should provide at least the following:

- *Abstraction* – the model should remove or hide all irrelevant detail so that the essentials stand out
- *Understandable* – the model should be expressed in a form that conveys the essential information directly and accurately (this implies that notation serves a critical function and should never be dismissed as just "syntactic sugar")
- *Accurate* – the model must correctly reflect the properties of interest of the modeled system to within the acceptable tolerances required
- *Predictive* – the model must be capable of accurately predicting the behavior and other properties of the modeled system (which, of course, depends on the accuracy of the model)

Models that are deficient in any of these categories are not likely to be useful. In case of software models and modeling languages of the past, models typically tended

to achieve high levels of abstraction, but more or less failed in the other three categories. In the following sections, we explain how these shortcomings can be overcome. First, we look at how modeling languages should be defined.

3.2 Languages for MDD

Model-driven development is an approach to software development in which models become essential artifacts of the development process, rather than merely serving an inessential supporting purpose.

The idea of MDD arose from early experiments with automatic code generation from computer-based software models. This has some obvious benefits: the code is an accurate realization of the model and its properties (assuming that the code generator is implemented correctly), it speeds up the process of programming, and it eliminates some of the tedium involved by automating the realization of common patterns.

To be truly useful in this way, it is necessary for the modeling languages to have semantics and syntax definitions that can be as precise and as accurate as those found in traditional programming languages. In this regard, some modern modeling languages, such as UML 2 (which was developed explicitly to support MDD), represent a major improvement over earlier generations of modeling languages [3].

With modeling languages of this type, it is possible to achieve very advanced and highly effective forms of MDD. In the most extreme form, it is even possible for such languages to cover the entire development cycle: from facilitating front-end design and analysis by providing high-level abstract models to serving as implementation languages. Note that one difference relative to programming languages is the ability for modeling languages to produce models that may be lacking low-level detail or that may be incomplete in parts, but which are still sufficiently formal to be used in design and analysis for making accurate predictions. (In contrast, most traditional programming languages require all details and all parts to be fully specified before they become formal artifacts.)

However, since modeling languages leave out implementation detail (which is the key to their value proposition), how can they also serve as implementation languages? Where is the information on the necessary low-level detail, detail that has to be specified in the implementation but which cannot be expressed using a modeling language?

This apparent dilemma leads many to the conclusion that, at some point in development, we must inevitably abandon the model and resort to traditional manual coding using programming languages. Given the modern preference for iterative and agile software development and the difficulty of maintaining the model and the code in perfect synchrony, the benefits of MDD for implementation seem limited to an initial power boost.

It does not have to be so. One common technique is to construct "heterogeneous models". In this type of model, the high-level aspects are captured using the modeling language while the more detailed aspects are specified using a detail-level language in the form of *fragments directly embedded in the appropriate parts of the model*. This approach combines multiple languages, each one suited to a different level of specification detail. For example, a state machine would be specified using the modeling language, most likely using a graphical syntax, whereas fine-grained details, such as the computations that take place in the course of a transition from one state to another, would be specified using a textual syntax. Detail-level languages that can be combined with modeling languages are called *action languages*, because they are mostly used for specifying detailed behavior (although they do not have to be limited to that). It is quite common to use an existing programming language as an action language. This has the added benefit of reusing available expertise, code libraries, and tools. The one difficulty is that such languages are not semantically aligned with modeling languages, with the possibility of writing actions that conflict with the semantics of the modeling language. However, in practice, this is much less of an issue than it might appear, because the embedded code fragments typically tend to be very small and relatively easy to inspect for such semantic violations. Alternatively, action languages have been defined that are semantically aligned with modeling languages – such as the actions semantics defined for UML 2 [3].

With this, it is indeed possible to extend the benefits of MDD to the full development cycle, including its use in agile development techniques based on multiple iterations. This means that, using MDD, it is possible to gradually *evolve* models into implementations without any major discontinuities in methods, expertise, or tools. This *continuum quality* seems unique to software. In all other engineering disciplines, the cross-over from models to actual implementations necessitates a change in materials, methods, tools, and expertise – discontinuities that often cause the desirable properties specified in the model to be lost or dissipated during implementation. With software, on the other hand, the model is an integral part of the implementation and can be automatically extracted from the final specification by simply applying a mechanized abstraction transformation on the resulting code.

3.3 The essence of MDD

If we examine what is at the core of MDD, we can easily identify two key themes:

- *Raising the level of abstraction* of specifications to be closer to the problem domain and further away from the implementation domain by using modeling languages with higher-level and better behaved constructs.
- *Raising the level of automation* by using computer technology to bridge the semantic gap between the specification (the model) and the implementation (the generated code).

The practical effects of these are higher product quality and increased productivity of development. This approach has already proven itself remarkably effective in the past with the introduction of so-called high-level languages in the late 1950's and the introduction of the compiler.

3.3 MDD and formal methods

Perhaps one of the most promising opportunities enabled by the new generation of modeling languages is the potential ability to analyze models by formal mathematical methods. It was noted earlier that this was infeasible for programs written using traditional programming languages because these languages usually contained low-level constructs that added intractable complexity. However, modeling language constructs can and should be defined with much more constraints, for instance, by basing them on well-known and tractable formalisms such as finite state machines or Petri nets. The good news here is that useful formal analysis techniques for these formalisms already exist. The introduction of well-behaved modeling languages with clear and precise semantics suddenly makes these techniques highly relevant.

The potential is to be able to predict *accurately* and with a high-degree of confidence the safety, correctness, and quality of service characteristics of a design in the early phases of development (when most of the critical design decisions are made). The higher levels of accuracy of such predictions stems from the continuum quality of software, which greatly increases the likelihood that the model properties are retained in the implementation.

As our society becomes more and more dependent on software for its everyday functioning, it is becoming increasingly more critical to ensure an efficient, predictable, and reliable and software design process.

At the time of writing, some progress has been made in exploiting this potential [4]. However, much more research and development needs to take place to adequately exploit this important potential.

4. Experience with MDD

MDD has been applied in many different forms. The most elementary is a basic one-time automatic code generation from a simple model. Once the code is generated, the model is typically abandoned or becomes a second-class concern. The next level up is so-called round-trip engineering, in which development alternates between work on the model and work on the code. This is probably the most common form of MDD, but it has the drawback that it does not really raise the level of abstraction significantly, since the concepts captured in the model are essentially constrained by the programming language. The more extreme and most productive form is fully-automated MDD, with the use of action languages to support implementation. In this mode, developers never directly deal with the auto-generated implementation code, but work exclusively with the model. This is analogous to the way that current programming languages are used.

Some of the projects using the more advanced forms are now over a decade old and represent large-scale mature software systems yielding millions of lines of auto-generated code developed by large development teams (cf. [5] [6]). Unfortunately, it is difficult to obtain precise data on such projects since they are often treated as confidential by the companies involved who believe that their use of MDD for such projects provides them with a major competitive advantage.

It may come as a surprise some to learn that many of the larger successful MDD projects are in the "technical" space (i.e., embedded and real-time software) [7], where one might expect exceptional sensitivity to overheads introduced by higher levels of abstraction and automatic code generation. However, many systems in this space tend to be extremely complex (since they have to cope with the unmitigated complexity of the physical world) and need the amplified expressive power that modeling languages provide. For example, the telecom domain has a long tradition of using complex finite state automata to specify system designs. Translating these into equivalent programs by manual means was highly error-prone and labor intensive. By using appropriate modeling languages and MDD tools, this manual step can now be circumvented, resulting in higher productivity and better product quality. Furthermore, the investment in MDD continues to pay back after the initial implementation since the model is much easier to understand and maintain than code.

5. The way forward

Although we now have substantial experience with MDD in its various forms, as described in the previous section, we are still in the infancy of this technological wave. Much technical work still remains to be done [8].

It is difficult to estimate the degree to which MDD is applied in practice, but it is quite clear that the vast majority of software developers and projects are still relying on more traditional development approaches. There are a number of reasons for this, some of them objective and some subjective.

In some cases, there is simply not enough awareness of what can and what has been achieved with the current generation of MDD methods and tools [9]. But, even when people are familiar with MDD and its potential, other factors may get in the way. One significant objective impediment may be the up front costs of introducing MDD into an established production environment. These include the costs of training staff and management as well as the cost of purchasing new MD tools. Another hurdle may be the lack of available expertise. For example, it is generally a good idea to include MDD experts on staff when it is first introduced into an environment. Unfortunately, such experts are still relatively rare and in high demand.

We should not ignore the problem of the relative immaturity of current MDD tools. Although much progress has been made in this regard, enabling the successes mentioned in the previous section, there is still a lot of opportunity for improvement. Many of the current generation of MDD tools are complex and have a steep learning curve. They are sometimes inflexible and often do not interwork with other complementary tools.

The emergence of industry-wide MDD standards such as those being defined by the OMG as part of its MDA initiative [10] will provide a major boost in this area. Among other things, it will enable greater specialization among toolmakers and tools, who can each focus on their area of expertise, relying on standards for interaction with complementary tools.

Perhaps the biggest technical issue at this point in time is the lack of a general theory of modeling language design. In contrast to programming languages, we still do not have any substantial theoretical underpinning for designing modeling languages and corresponding tools. Tools already exist that allow users to define domain-specific modeling languages, but without a proper theory on how such languages should be defined, this can be as much a liability as it is an opportunity. .

In terms of subjective impediments, the biggest by far is the "culture" factor. That is, many software developers are unprepared to make the jump to MDD. While some of this push-back is based on realistic pragmatic concerns described earlier, much if not most of it can be traced to simple human resistance to change. After all, software practitioners invest much time and effort mastering the current generation of programming technologies and it is perfectly understandable that this is not an investment one sheds lightly. Part of the problem is that many software developers view themselves as experts in particular programming technologies, as opposed to experts in the domain for which they are writing the software. For example, programmers working on communications software often do not know much about communications technology. Consequently, proposed changes to the underlying technology, no matter how much more productive it might be, is met with stiff resistance. As might be expected, if the development team is not open to adopting the new technology this can be quite serious for a project. Therefore, the introduction of MDD has to be handled carefully and with great consideration to the human aspects.

5. Summary

The conceptual framework for the dominant programming technologies of today was created almost fifty years ago, when the demands placed on software were far less challenging than they are at present. This outdated framework is characterized by an excess of accidental complexity that compounds the already difficult problem of developing reliable software reliably.

MDD is an approach to software development that aims at removing much of this unnecessary complexity, by combining the effects of more direct expression of design solutions with higher-level languages and the intensified use of computer-based automation.

Experience with MDD over the past decade or so indicates that the supporting technology has reached a level of maturity in which it can be applied successfully to significant industrial projects, yielding the anticipated boosts to productivity and quality.

Nevertheless, we are still in the pioneering phase of this new approach and much practical and theoretical development needs to occur before MDD becomes a dominant style of development.

References

[1] F. Brooks, *The Mythical Man-Month,* 2nd ed., Addison Wesley Longman, 1995.

[2] M. Fowler, *UML Distilled*, Addison Wesley, 2005.

[3] Object Management Group, *Unified Modeling Language: Superstructure*, Version 2.0, OMG document formal/05-07-04, 2005.

[4] S. Baranov, et al., "Leveraging UML to Deliver Correct Telecom Applications," Chapter 15 in L. Lavagno, et al. (eds.), *UML for Real: Design of Embedded Real-Time Systems,* Kluwer Academic Publishers.

[5] V. Kulkarni and S. Reddy, "Model-driven development of enterprise applications", in N. Nunes et al. (eds.), *UML Modeling Languages and Applications,* (LNCS 3297), Springer, 2005.

[6] C. Raistrick, "Applying MDA and UML in the Development of a Healthcare System", in N. Nunes et al. (eds.), *UML*

Modeling Languages and Applications, (LNCS 3297), Springer, 2005.
[7] L. Lavagno, G. Martin, and B. Selic, *UML for Real: Design of Embedded Real-Time Systems,* Kluwer Academic Publishers.
[8] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, September/October 2003, IEEE Computer Society, 2004.
[9] N. Nunes et al. (eds.), *UML Modeling Languages and Applications,* (LNCS 3297), Springer, 2005.
[10] Object Management Group, *MDA Guide*, Version 1.0.1, OMG document omg/03-06-01, 2005.

Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing
0-7695-2561-X/06 $20.00 © 2006 IEEE