Autom Softw Eng (2008) 15: 379–391
DOI 10.1007/s10515-008-0035-7

Personal reflections on automation, programming culture, and model-based software engineering

Bran Selic

Received: 21 July 2008 / Accepted: 1 August 2008 / Published online: 3 September 2008

Abstract Model-based software engineering (MBSE) is an approach to software development characterized in part by significantly greater levels of automation when compared to more traditional development methods. Computer-based tools play a fundamental role in a number of key aspects of development, including authoring support (many MBSE languages are predominantly visual), automatic or semi-automatic verification, automated translation of specifications into corresponding programs, and so on.

Given the historical precedents, such as the introduction of compilation technology, there is little doubt that automation, when properly conceived and realized, can dramatically increase the productivity of software developers and improve the quality of their software. Therefore, it is natural to assume that MBSE would quickly become the dominant form of software development, similar to the rapid adoption of computer-aided design approaches for hardware. Yet, this has not been the case.

In this opinion-based article, derived from the author's long-term experience with MBSE and its application in industry, we examine the causes behind this seemingly paradoxical situation.

Keywords Model-driven development · Computer-aided software engineering · Psychology of programming · Usability · Computer automation · Software tools

1 Introduction

The use of computer technology in support of software design and development dates back to the earliest days of programming. It is an obvious fit, given that the computer is, in effect, the ultimate automation machine and that the primary artifacts of the

B. Selic (✉)
Malina Software Corp., 10 Blueridge Court, Nepean, Ontario, Canada K2J 2J3
e-mail: selic@acm.org

development process are themselves stored, manipulated, and executed on computers. Computer-based automation has been used for a variety of functions related to software development including compilation, program linking and loading, source program creation and editing, version management, debugging, verification, documentation, and so on. Of these, perhaps the most significant in terms of greatest impact on productivity and quality, is compilation. The introduction of compilers enabled so-called *high-level* programming languages (also referred to sometimes as *third-generation languages*), which reduced the complexity involved in program design by eliminating the need for program writers to concern themselves with many technology-specific details. This not only made it possible to port a given program to a different machine with little or no modification, but, more importantly, it also allowed programs to be specified using concepts and constructs that were much closer to human understanding and to the problem domain.

These important benefits were quickly recognized and the vast majority of practitioners switched from low-level to high-level language programming in a relatively short period of time. Furthermore, programming became more approachable, leading to an increase in the number of both programmers and applications.

Still, as demand grew for ever more complex and diverse computer applications, the abstraction level provided by common third-generation programming languages such as C, Fortran, or Basic proved inadequate despite numerous incremental improvements to these languages. Specifically, the basic constructs provided by these languages were often found to be too fine grained to allow direct and clear expression of the more complex and domain-specific concepts and relationships of many software applications (i.e., their architecture).

This naturally led to a further elevation of the abstraction level of software specifications, through use of higher-order[1] formalisms, such as finite state machines or entity-relationship structures. (It is worth noting here that many of these formalisms were inherently graphical in nature, since graphical renderings are often more effective in describing certain types of structures and relationships compared to text.) The use of such higher-level formalisms has now become standard practice in the analysis and design of complex software systems.

However, although there is obvious similarity here to the shift in abstraction levels that took place during the switch from second- to third-generation languages and despite the fact that higher-level implementation languages, such as ROOM (Selic et al. 1994), Statemate (Harel et al. 1990), or SDL (Ellsberger et al. 1997), have been around for decades, there has not been a comparable massive adoption of more modern *implementation* practices and technologies. This has resulted in a widening semantic gap between software design specifications, which are typically expressed using higher-level formalisms, and their corresponding implementations, which are usually specified using third-generation programming languages. Although there have been numerous advances in programming languages over the past three decades, the essential level of abstraction (and, consequently, the expressive power)

[1] What makes them higher order is that they express problem domain concepts more directly and more succinctly and abstract out even more of the underlying implementation technology.

of today's dominant implementation languages, such as Java, C++, or C#, is not significantly greater than that of the earliest third-generation languages. That is, it is almost as difficult to discern the high-level architectural form and key design principles of a program written in Java as it would be if the same program were written in Fortran or Cobol.

Needless to say, this discrepancy in abstraction causes problems that are often very difficult to overcome. There is the obvious risk that, in the process of informally translating high-level specifications into programs, errors will be introduced such that the implementation does not accurately capture design intent. On the other hand, it is also possible that, in ignoring key implementation concerns, a high-level design specification could prescribe inefficient and even infeasible systems. To minimize the likelihood of such pitfalls, complex software systems are best designed through some type of iterative and incremental process, in which analysis, design, and implementation activities proceed either in parallel or cyclically follow each other in quick succession. The wider the abstraction gap between the concepts used in these activities the more difficult it is to iterate between them.

The use of computer-based automation to assist in bridging the semantic gaps in this process seems like an obvious choice. In fact, there have been numerous initiatives to introduce such automation into software development, starting with so-called *fourth-generation languages*, through computer-aided software engineering (CASE) tools, to the current model-based software engineering (MBSE). Let us examine each of these in turn.

2 Fourth-generation computer languages

These languages (often referred to as *4GL*) are high-level languages specialized for a specific application domain or purpose. An early example of a 4GL is the Report Program Generator (RPG) language (International Business Machines 1964), a declarative language used to generate reports from databases. It proved highly effective for its intended purpose, greatly reducing development time compared to equivalent imperative-style programs written in a language such as Cobol. Many other 4GLs, including notably SQL (for database queries) (Chamberlin and Boyce 1974), MATLAB (for mathematical and dynamic system modeling) (MathWorks 2008), and SPSS (for statistical analysis) (SPSS Inc 2006), have been defined and proven successful over time. The current focus on domain-specific (modeling) languages (DSLs) is simply the latest manifestation of this trend (Greenfield et al. 2004; Mernik et al. 2005).

The primary drawback of 4GLs stems from the very same characteristic that makes them successful: their highly-specialized nature. Because they have a limited scope of application, these languages appeal to a limited user base. This usually implies that it much less cost effective to build highly sophisticated automation support for such languages compared to the much more common general-purpose programming languages. More often than not, tools to support these languages are either custom built in-house tools or provided by a small number of specialized vendors (often just one). In-house tools mean dedicating development and other resources to tool support

and evolution, resources that come at the expense of core business. On the other hand, tools that are supported by a small number of vendors are typically more expensive and often carry a high risk that either the vendor or the tools may be discontinued. Furthermore, highly specialized tools and languages require specialized training and skills that are more difficult to come by on the open market, implying higher training costs.

In contrast, there is a plethora of excellent and diverse development tools that support common general purpose programming languages. These tools are produced and supported by both vendors who specialize in such tools as well as by the open source movement. Competitive pressures as well as the altruistic motives behind open source ensure that these tools are constantly improving while their cost keeps dropping (many are available for free). The economies of scale simply do not allow an equivalent situation to develop for 4GLs. (This is a factor that is unfortunately often overlooked in recent discussions about the advantages of DSLs.)

3 CASE tools

CASE tools were inspired by the undeniable success of *computer-aided design (CAD)* tools, which introduced high levels of automation in hardware design. CASE tools represent an early initiative to directly translate higher-level formalisms used in analysis and design of software systems into equivalent code—an idea that seems beyond reproach. They not only helped with the production of various analysis and design diagrams but also typically supported some form of automated or semi-automated code generation from them. Unfortunately, CASE tools never came close to matching the success of their hardware counterparts. In fact, they are now frequently cited as a paradigmatic example of yet another technology that promised a breakthrough which it failed to deliver.

There are several reasons behind the "failure"[2] of CASE that are worth examining in detail, since they hold lessons to be learned for any current or future attempts to bridge the gap between design and implementation by means of computer-based automation.

One prominent obstacle lies in the qualitative differences between design and implementation. Design, particularly in its early phases, is best done in unconstrained circumstances, where ideas can form, unfettered by the constraints of pedantic precision and formality. Consequently, most of the design languages supported by CASE tools tended to support formalisms that were imprecise and informal. Implementation languages, on the other hand, must necessarily be unambiguous and highly formal since they have to be translated deterministically and precisely into deterministic and precise program code. Therefore, any program code generated from such formalisms were incomplete and needed to be supplemented with programmer-written code, that removed any ambiguity and also added missing implementation-level detail. This, unfortunately, breaks the formal link between the code and the specification from

[2]The quotation marks are there since, in my opinion, CASE was not a failure; these tools were a necessary but insufficient step towards more modern and more effective technologies based on the same premises.

which it was derived. The added code can subvert or even contradict the original design goals either by accident or by intent. At that point, it may no longer be possible to automatically revert to the high-level formalism—thereby impeding or even blocking further iterative development.

There have been numerous attempts to get around this problem using a process known as *round-trip engineering (RTE)*. In this process, the code is reverse engineered into a corresponding high-level specification. However, the most common outcome of this process is gradual deterioration of the high-level specification to a one-to-one graphical representation of the low-level code.[3] In such cases, much of the value stemming from a high-level representation is eroded.

Furthermore, the code that was generated by CASE tools was often either trivial (thereby providing little true value to developers) or of poor quality compared to hand-crafted code. Part of this is due to the fact that there was little or no theoretical understanding of or experience with how to best generate code from graphical formalisms.

Another problem with early CASE tools was due to the overabundance of different high-level formalisms that emerged at the time when CASE tools were being introduced. For example, in the early 90's, there were close to a hundred published analysis and design languages (Graham 2001). Individual CASE tools would typically support some subset of these, but rarely more than a few. This forced users into the vendor lock-in problems already discussed for 4GLs.

In summary, CASE tools simply did not provide enough value to either the designers or the implementers to justify more widespread dedicated use. Most CASE tool vendors disappeared or were absorbed by other vendors with a broader perspective on model-based software engineering (MBSE).

4 Model-based[4] software engineering (MBSE)

In a sense, MBSE is simply a continuation of the CASE approach. However, there has been a significant context shift since the early days of CASE, which makes MBSE significantly more viable. Specifically:

- There has been considerable progress in the underlying technologies. This includes, notably, more powerful computing hardware (performance, memory capacity) as well as advances in modeling language design (the use of meta-modeling approaches), automated code generation methods, and software tooling (the introduction of tool frameworks such as Eclipse (Eclipse Foundation 2008)). In general,

[3]To be precise, RTE is still used in many places, but primarily as an implementation assist facility rather than a high-level analysis and design tool.

[4]The terms *model-driven development* or *model-driven engineering* are more commonly encountered. However, I feel they are rather misleading. Namely, even though models play a first-order role in this approach, they do not really drive development, but are merely a by-product of the development process (as they are in any engineering process). Furthermore, there is a danger that, through such narrow labelling, other equally important aspects and styles of development (such as test-driven development, requirements-driven development, etc.) may be neglected or, worse, deemed as incompatible rather than complementary.

as our experience with the design and use of modeling languages grows there is a corresponding increase in our understanding of both the solutions and the problems that accompany them. (Still, it is fair to say that MBSE is still far from being an established engineering discipline, that is, one based on well-understood scientific and technical foundations.)

- The emergence and widespread adoption of a number of key industry standards, most notably the Object Management Group's Unified Modeling Language (UML) (Object Management Group 2007a), which has greatly reduced the problem of the gratuitous profusion of different high-level modeling languages and notations.

Much has been written about MBSE and what characterizes it (e.g., Object Management Group 2003; Frankel 2003; Greenfield et al. 2004; Mellor et al. 2004). There is a lot of discussion about platform-independent models (PIMs) and platform-specific models (PSMs), meta-modeling, domain-specific modeling languages, model transforms, etc. In this author's opinion, the core idea of MBSE can be reduced to two fundamental notions that we have already discussed:

1. *Raising of the level of abstraction;* that is, raising the level of software specifications even further away from underlying implementation technologies (relative to, say, traditional programming languages) and
2. *Raising the degree of computer-based automation* used to bridge the widening gap between design specifications and corresponding implementations.

In most engineering practice, the term "model" is used to denote an abstract representation of some concrete engineering or other artifact—something that abstracts out uninteresting detail. This could be a mathematical model or a scale model, or some other type of model, but in all cases it is distinct from the real-world entity that it represents. However, in the context of MBSE, "model" is often used as a generic term to denote any specification expressed using a higher-level formalism, whether it is an abstraction that omits detail or a fully-fledged implementation specification from which a complete executable program can be auto-generated. This peculiar practice can be traced to the unique nature of MBSE, in which the final development artifact can, in principle, be the result of a series of incremental refinements of successive high-level specifications. In the course of this process, the same language, tools, medium, methods, and expertise can be used throughout, thereby avoiding the qualitative discontinuities that characterize practically every other form of engineering development. Clearly, such a process, if properly supported by automation, has a much greater likelihood of ensuring preservation of the original design intent. Furthermore, with suitable computer-automated transforms, it is always possible to reduce a fully-detailed implementation model into a more abstract form which is easier to comprehend and which makes it easier to detect any unintended or undesirable design modifications.

Note that, during the early phases of this continuous process, it is useful to keep the level of precision and formality relatively low, allowing a freer (and looser) expression of design ideas. As the process progresses, the degree of formality and consistency checking should be increased correspondingly, until, in the final phases, it is equivalent to the degree of formality associated with programming languages. This requirement to support progressively increasing levels of formality is one important feature that distinguishes good modeling languages from most programming

languages. This same capability also differentiates good MBSE design tools from traditional programming tools.

5 Pragmatic issues with computer automation in MBSE

It is evident from the above description that MBSE is not practical without effective computer-based automation. And, although there are numerous examples of successful applications of MBSE in large industrial software projects, realized using the current generation of MBSE tools (cf. Weigert and Weil 2006; Nunes et al. 2005), the state of the art of MBSE tools leaves much to be desired. In particular, it is my opinion that most MBSE tools suffer from a number of serious deficiencies:

- *Usability problems.* The key issue here is the accidental[5] complexity of these tools. Undoubtedly, the infrastructure required to support MBSE, with its innovative but uncommon graphical languages, sophisticated model transforms, automatic code generation capabilities, etc., is inherently more complex than the infrastructure required for traditional text-based programming languages. It requires more effort to set up and tune to a particular production environment requires significant effort. However, on top of this essential complexity, the vast majority of current MBSE tools adds gratuitous complexity and provides mostly token support to help users cope. Thus, the interfaces of these tools are generally not based on any deep study of standard usage patterns or expert knowledge of human psychology. The typical MBSE design tool offers its capabilities via multiple overlapping categories of menu items grouped in unintuitive ways. In an often naive and simplistic interpretation of principles of GUI design, garnered mostly through ad hoc learning rather than systematic study, bizarre and confusing icons and graphics abound, supposedly providing an intuitive interface, but more often achieving the opposite effect. For users, understanding what such a tool can do and how to do it requires major expenditures of time and effort—time and effort that should have been more valuably expended on solving the application problem at hand. Thus, tools, which are intended to boost productivity, can actually reduce it.

  My perception from observing development teams is that usability is still a second-order concern in the design of the vast majority of software tools. It is often incorrectly interpreted as merely a matter of providing a "fancy" user interface.[6] Therefore, usability experts, if consulted at all, are typically asked to comment and advise on the look and feel of a tool's interface long after the architecture of a tool has been set.

[5] In his insightful book on the nature of software, *The Mythical Man-Month* (Brooks 1995), Fred Brooks Jr., categorizes *accidental* complexity as that which is not an essential part of a problem but is the result of artificial barriers imposed due to lack of foresight or reflection.

[6] Of course, user interface design is an important aspect of usability, but certainly not the only one. Usability has to do with how a tool interacts with its users, which implies that it must be figure in the tool's architecture.

One promising approach to dealing with this problem might be to use an *intelligent adaptation* approach in which tools dynamically adapt themselves and their interfaces to users and their usage patterns (Magerko 2008). This type of usability approach can be found in some game-playing programs, which start off with a basic set of capabilities and then gradually expose more and more of their capabilities as users become more sophisticated and as data is gathered on usage patterns.

Another manifestation of the lack usability in today's MBSE tools is their inadequate support for customization for specific application domains and environments—this despite the presence of numerous configuration options found in most tools. However, these options are typically limited to a set of choices defined by the tool's designers, who, as noted earlier, often have an inadequate understanding of the application domain or how the tool is to be used. Furthermore, custom configurations are defined for individual tools independently of other tools in the same tool chain, making it difficult to ensure consistency of customizations across tools.

- *Interoperability problems*. This refers not only to the fact that, despite the existence of model interchange standards such as XMI (Object Management Group 2007b), it is rarely possible to effectively exchange models between equivalent tools from different vendors, but also to the inability to exchange models between complementary tools. For example, it may be required to transfer a model from a model authoring tool to a specialized analysis tool where it can be analyzed for certain properties (safety, liveness, performance characteristics, etc.). Unfortunately, in most cases this transfer is fraught with problems and requires some intervention. There are two reasons for this. First, the interchange standards themselves are not precise enough to ensure an accurate model transfer, that is, a transfer without loss of key information. There are ambiguities in how a model is serialized into a textual form (for transfer) so that it is interpreted correctly and fully in the receiving tool. Second, the tool vendors have so far shown little inclination to fix the problems in the interchange standards. This is not surprising, since they have a vested interest in keeping their customers bound to their products rather than their competitors' products. However, part of the fault here lies with the customers themselves, who, although they often complain about this state of affairs, rarely exert significant pressure on vendors to fix the problem. Until that happens, interoperability problems will remain.
- *Scalability problems.* The abstraction power of models is needed most when dealing with large and complex systems. As models of such systems progress through successive refinements that add more and more detail and, as more and more individuals get involved in working on the model, the amount of information that needs to be maintained increases significantly. A crucial part of this information is the internal structural relationships that capture the semantic linkages between different parts of the model (or between different models). They are indispensible when querying the model (e.g., to assess the impact of a change to the design). In effect, an MBSE model is a complex network of interconnected elements in which more or less everything is connected (directly or transitively) to everything else.[7]

[7] It can be justifiably argued that the complexity of semantic linkages of a system is independent of whether or not it is specified as a model or as a program. However, in text-based specifications, many semantic

This, of course, makes it extremely difficult to partition such a model into manageable units that may evolve in parallel. To get around this problem many tools require the full model to be loaded before it can be manipulated, which hampers their ability to deal with large models.

In summary, the tooling problems cited above and the issues of usability in particular present major impediments to a broader application of MBSE and thus discourage many who are interested in taking advantage of its benefits. However, there are some additional factors related to MBSE that may present even greater hurdles. These are discussed next.

6 On the influence of programming culture on MBSE

"Problems cannot be solved by the same level of thinking that created them."
(Albert Einstein)

Vendors of MBSE tools often say that the *status quo* is the single most significant issue blocking the broader adoption of MBSE in practice. By this they mean the pervasive culture and psychology of traditional programming practice.

The source of this problem can be traced to the rather unique nature of programming and to the type of personality that is attracted to it. One key element that distinguishes programming from other forms of engineering design is its lack of physical impedance. That is, programming primarily involves the transfer of ideas into equivalent or near-equivalent specifications and does not require bending, lifting, or otherwise processing of physical materials nor does it involve protracted manufacturing and assembly. The main ingredient involved in software production is information. With no appreciable physical effort involved, the delay from idea to its realization (in the form of a compiled and executing program) can be in the order of a few minutes if not seconds. This is quite exceptional in engineering practice, where the prove-in of a design idea typically requires months or even years and involves painstaking and protracted analysis and design.

While this rapid turnaround is an obvious benefit, it does have some important consequences whose effects, on reflection, are not necessarily positive. One of these is that it often creates an impatient state of mind that discourages reflection. The inertia that is inherent in traditional engineering design, where the time cost of bad design decisions can be prohibitive, necessitates that design be a highly thorough and systematically organized process. It requires a deep and lengthy analysis of possible consequences of key design decisions that often leads to better understanding of the issues and more optimal solutions.[8] Unless strong discipline is enforced, software design often bypasses this reflective phase; many solutions are hacked by successive

linkages are only established during compilation or link time, which means that our ability to query such specifications is limited. But, one of the touted advantages of model-based specifications is that such linkages are present in the model and that they can be queried at any time. Therefore, there are greater expectations for model-based specifications.

[8] As a brilliant engineer friend of mine once said: "If you think about a problem long enough, you will always find a better solution for it."

minor modifications of an inadequate initial design concept until the desired output is finally achieved—usually a highly suboptimal one.

While this lack of what is sometimes called *system-level thinking* in software is clearly a problem, there is an even deeper issue lurking behind this unique inertia-less property of software. The ability to conceive designs and have them confirmed by a running program in a short interval of time is a particularly satisfying and highly seductive experience. For many individuals, the sense of personal gratification and mastery that comes when a program executes successfully is so appealing that it leads to a kind of infatuation with programming that can be highly addictive. A common and unfortunate consequence of this phenomenon is that in many programmers' minds the focus shifts from the system being constructed to the process of programming; a specific manifestation of the now familiar "the medium is the message" syndrome first described by the philosopher Marshall McLuhan (1964). One common undesirable consequence of this is loss of focus resulting in an insufficient understanding of and concern for the product being built. Such individuals—and I believe they constitute a significant proportion of software practitioners—identify themselves primarily by the programming skills that they have mastered (after investing significant time and effort) and not by the types of systems that they help construct. Thus, they do not view themselves as, say, financial system experts or embedded systems experts with programming skills, but, instead, as Java experts or C++ experts, Linux experts, etc. Their sense is that they are equally competent to work on any type of system, as long as it takes advantage of their particular technological skills. An analogy to this might be someone who is an expert riveter, who can work on any project that requires riveting, whether it is an ocean liner, airplane, or a skyscraper—it does not matter. Generally, one does not expect riveters to advocate newer technologies that might displace riveting or, for that matter, to fret over the purpose and architecture of the system they are helping construct.

So, how does this stand in the way of greater propagation of MBSE in practice?

The difficulty lies in that programmers of this type, with little or no interest in the end product or its usage, are often unwilling to switch to new technologies that take them out of their comfort zone, even in cases where such technologies might be much better suited to the problem on hand. Therefore, the combination of new languages and tools required for MBSE are viewed as a threat. It seems rather ironic that it is these individuals, who work with the most advanced technology ever devised, who are prone to be so highly conservative. This attitude can be contrasted with the exceptionally rapid adoption of a similar technology (CAD) by hardware designers, who, as noted earlier, saw it as an opportunity to build end products much more effectively. Compounding this problem is the sheer number of such classically trained software professionals, numbering in the millions. This is a significant inertial mass that will likely keep impeding broader adoption of MBSE.

7 Additional opportunities for automation in MBSE

In addition to the ability to automate model creation and code generation, MBSE offers other enticing opportunities to take advantage of computer-based automation

during development. In this section, we briefly discuss two of the most promising ones:

- *Formal computer-based design analysis.* The problem of practical formal checking of the safety and liveness properties of a software program has been greatly hindered by the highly complex nature of the dominant programming languages. The semantics of these languages are so intricate that their corresponding mathematical models used in analysis are extremely complex and invariably lead to scalability issues (e.g., the well-known state-explosion problem). The opportunity created by MBSE is the possibility of defining a new generation of computer languages. These new modeling languages can be based on less complex formalisms, such as state machines or Petri nets, which are much more open to formal analysis than programming languages.

  Note that this type of computer-assisted analysis can also be extended to analyzing not just the qualitative properties of a design, such as absence or presence of deadlocks, but also its quantitative aspects. For example, if a design model is annotated with suitable performance-related information (e.g., worst case execution times, deadlines, throughput rates, etc.) it is possible to analyze such a model using modern performance analysis techniques (e.g., based on queuing theory) to determine its time-related characteristics. This can be achieved by transforming the original model into a model suited to performance analysis, such as a queuing model, which is then analyzed by a specialized performance analysis tool. There are practical examples of the viability of this approach based on the MARTE profile of UML (Object Management Group 2008a, 2008b). Other types of quantitative property analyses are possible as well, including timing analysis, security analysis, availability analysis, etc. By automating the transformations from one type of model to another and using computer-based analysis techniques, the need for scarce analysis expertise can be minimized or even eliminated.
- *Model simulation.* The ability to translate modeling language specifications into equivalent computer programs means that the modeling languages must have precise semantics. This, in turn, implies that specifications specified in such languages can also be executable. "Executable" generally means that the specification can, in principle, be executed on a virtual machine that directly interprets the modeling language. This ability is important for a number of reasons. The primary one is that it becomes possible to do a practical evaluation of the validity of a proposed design by executing it on a computer.

  The value of such an evaluation is greater if it is performed on a very abstract version of the model, before too much effort and resources are expended on an inappropriate design choice. This in turn implies the ability to execute very high-level, abstract models; that is, models that have the high-level features that are being evaluated specified sufficiently but little else. Such a capability is not as complicated to provide as it might first appear: when an ambiguity in the specification is encountered, the model execution system may ask for external (e.g., human) guidance on how to proceed or it may use certain pre-configured assumptions of its own.

  One interesting and significant side effect of this type of early execution is the confidence boost that a design team gets from seeing something work early in the

development cycle. The value of this cannot be overestimated, particularly when dealing with sophisticated software architectures, where the initial degree of confidence is low. In fact, one of the main reasons why many practitioners dislike models in favor of programming is the lack of such positive re-enforcement during development. Waiting until the very end to determine whether or not a design is viable is not only risky, it also very stressful. Having executing versions of a design in the earliest phases of development will both reduce risk and alleviate the stress.

8 What the future holds

Despite all the shortcomings of current tooling, MBSE has proven its viability and value in numerous industrial applications. However, to reach its full potential, major additional breakthroughs still need to happen. In my view, these need to occur in two principal directions:

1. We need to evolve a systematic theoretical understanding of the various key capabilities that are at the core of MBSE, such as the principles of modeling language design, model transformations, code generation, automated verification, and so on. At present, there are many excellent ideas and methods in these areas, contributed by both industry and research, but they are still not sufficiently understood. Thus, committing to MBSE in practice still requires a great deal of improvisation, invention, and experimentation and still carries with it significant risk. The objective must be to transform it into a reliable and well understood engineering discipline.
2. Significant improvements must be made in computer-based automation, which means mitigating and overcoming all the technical challenges described earlier (usability, interoperability, and scalability). Undoubtedly, some of these will be much easier to achieve once a proper theoretical foundation is in place.

As for the cultural factor, one can only hope that the cumulative effect of continuing successes of MBSE-based projects will eventually create sufficient critical mass to propel the substantial community of recalcitrant developers to be more open to the new technologies. It is my belief that, part of the key here lies in developing computer-based automation that is elegant and sophisticated without being intimidating. The potential is there, it is time to use it.

References

Brooks Jr., F.: The Mythical Man-Month. Addison-Wesley, Reading (1995). Anniversary edn.

Chamberlin, D.D., Boyce, R.F.: (1974). SEQUEL: a structured English query language. In: Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control, pp. 249–264. Association for Computing Machinery (1974)

Eclipse Foundation: Eclipse documentation. http://www.eclipse.org/documentation/ (2008)

Ellsberger, J., et al.: SDL – Formal Object-Oriented Language for Communicating Systems. Prentice Hall, London (1997)

Frankel, D.: Model Driven Architecture – Applying MDA to Enterprise Computing. OMG Press, Indianapolis (2003)

Graham, I.: Object-Oriented Methods. Addison-Wesley, London (2001)

Greenfield, J., et al.: Software Factories. Wiley, Indianapolis (2004)

Harel, D., et al.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Transactions of Software Engineering **16**(4), 403–414 (1990)
International Business Machines (IBM): Systems reference Library: Report Program Generator (on Disk) Specifications. http://bitsavers.org/pdf/ibm/14xx/C24-3261-1_1401_diskRPG.pdf (1964)
Magerko, B.: Adaptation in digital games. IEEE Computer **41**(6), 87–89 (2008)
MathWorks: MATLAB Function Reference. http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html (2008)
Mellor, S., et al.: MDA Distilled—Principles of Model-Driven Architecture. Addison-Wesley, Boston (2004)
Mernik, M., Heering, J., Sloane, M.: When and how to develop domain-specific languages. ACM Computing Surveys **37**(4), 316–344 (2005)
McLuhan, M.: Understanding Media: The Extensions of Man. McGraw-Hill, New York (1964)
Nunes, N.J., et al. (eds.): Industry track papers. In: UML Modeling Languages and Applications – «UML» 2004 Satellite Activities, Lisbon, Portugal, October 2004 (Revised Selected Papers). Lecture Notes in Computer Science, vol. 3297, pp. 94–233. Springer (2005)
Object Management Group (OMG): MDA Guide, v.1.0.1. OMG document omg/2003-06-01 (2003)
Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, v.2.1.2. OMG document formal/07-11-02 (2007a)
Object Management Group (OMG): XML Metadata Interchange (XMI), v.2.1.1. OMG document formal/07-12-01 (2007b)
Object Management Group (OMG): A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, v.Beta 2. OMG document ptc/08-06-09 (2008a)
Object Management Group (OMG): OMG MARTE Information Day (June 2008b). http://omgmarte.org/Events.htm
Selic, B., et al.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)
SPSS Inc: SPSS 15.0 Command Syntax Reference, Chicago IL (2006)
Weigert, T., Weil, F.: Practical experience in using model-driven engineering to develop trustworthy computing systems. In: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, vol. 1, pp. 208–217, 5–7 June, 2006