Softw Syst Model (2012) 11:513–526
DOI 10.1007/s10270-012-0261-0

EXPERT'S VOICE

What will it take? A view on adoption of model-based methods in practice

Bran Selic

Received: 28 December 2011 / Revised: 18 May 2012 / Accepted: 9 July 2012 / Published online: 10 August 2012

Abstract Model-based engineering (MBE) has been touted as a new and substantively different approach to software development, characterized by higher levels of abstraction and automation compared to traditional methods. Despite the availability of published verifiable evidence that it can significantly boost both developer productivity and product quality in industrial projects, adoption of this approach has been surprisingly slow. In this article, we review the causes behind this, both technical and non-technical, and outline what needs to happen for MBE to become a reliable mainstream approach to software development.

Keyword Model-based engineering

1 Introduction

> "So that we may say the door is now opened, for the first time, to a new method fraught with numerous and wonderful results, which in future years will command the attention of other minds".
>
> – Galileo Galilei, sixteenth century

> "What is not fully understood is not possessed."
>
> – Johann Wolfgang Goethe, German author and philosopher

Traditional engineering disciplines have been making use of models and modeling from its earliest days (e.g., [36]). Engineering models serve a number of key purposes in design: they help engineers reason and make predictions about both problems and solutions, they provide an effective means for communications between different stakeholders, and they often serve as specifications (i.e., "blueprints") for implementation. In fact, it is safe to say that much of the undeniable success of modern engineering can be attributed to the proficient use of models of various kinds (mathematical, scale, etc.).

Software designers and developers have also been using models practically from the very beginning of the discipline with techniques such as flowcharts and finite state machines. However, it seems that in software design models rarely play the same central role as they do in traditional engineering. If they are used at all—and many software practitioners prefer to shun them entirely—they are often perceived as secondary, inessential, artifacts that merely complicate matters. Yet, numerous industrial experiences with development approaches grouped under the general category of *model-based engineering* (MBE), have clearly demonstrated that models can improve both the quality of software and the productivity of teams that develop it (e.g., [5,15,37]). Nevertheless, MBE is far from being a dominant software development paradigm and is still seen by many practitioners as either unproven or, even worse, as yet another waning fad in a discipline that already suffers from an excess of silver bullets.

When the editors of this issue asked me to set down my views on what needs to happen for MBE to be accepted as a full-fledged trustworthy technical[1] discipline, my first inclination was to interpret this as a technical question; that is, as a question about necessary technological and methodological advances. Only after some reflection did it dawn on me that

Communicated by Prof. Jon Whittle and Gregor Engels.

B. Selic (✉)
Malina Software Corp, Nepean, ON, Canada
e-mail: selic@acm.org

[1] I eschew here the distracting and often unproductive controversy as to whether or not software development is a form of "engineering" and shall refer to it throughout as a "technical" discipline.

this line of thinking was actually symptomatic of one of the main hurdles that stands in the way of broader penetration of MBE. Namely, being a technology developer by profession, I was seeking a purely technical solution to a problem that cannot be solved by technology alone. As I have argued in a previous article [32] and as borne out by recent systematic studies of the use of MBE in industrial settings [5,14,15,23], a number of complex social and economic issues are at play here, in addition to the technical ones. In fact, based on long-term experience in industry, it is my opinion that these non-technical issues are the more critical ones to overcome.

A common response by many of my professional colleagues to this type of situation is that, given that we are not sociologists, cultural anthropologists, or politicians, our only recourse is to contribute to the technological aspects of a solution, leaving other aspects to those more qualified. This view, again influenced by a traditional technologist thinking (i.e., separation of concerns), seeks to isolate technology from its socio-cultural context. But, how can we claim that our solutions are appropriate if we ignore what are perhaps the most crucial aspects of almost any engineering problem: the bottom-line effectiveness of technological solutions in the overall social context in which they are used? Albert Einstein is purported to have said: "*Concern for man himself and his fate must always constitute the chief objective of all technological endeavors…in order that the creations or our minds shall be a blessing and not a curse to mankind. Never forget this in the midst of your diagrams and equations*" [9].

Unfortunately, it has been my experience that the tendency to ignore the greater socio-economic context in which their solutions operate is more prevalent in software development than in other technical disciplines. Application knowledge is often fragmented and superficial. Part of this can be ascribed to the highly seductive nature of software, which has the unique and highly gratifying property that the cycle from conception to realization can be measured in minutes.[2] Consequently, it tends to attract practitioners who are more focused on the technology itself than the application they are building with that technology—a particular case of Marshall McLuhan's "the medium is the message" syndrome [18]. Like the proverbial "man with a hammer" who sees all problems as nails, they seek to solve problems using the particular computing technology with which they are familiar (e.g., Java, C++, Linux, Android, etc.), as opposed to looking for solutions that best fit the application on hand. From that perspective MBE technologies are disruptive, since one of the primary MBE objectives is to abstract away as much of the underlying computing technology as possible, removing thus the need for a deeper understanding of it. Consequently, MBE is perceived as a threat by such people and, as a result, its failures are often welcomed and magnified while its successes are questioned.

Overcoming this Luddite-like phenomenon is complicated by the fact that there is a very large number of individuals whose primary source of income comes from writing software. Although there do not seem to be any trustworthy worldwide statistics, the number of software practitioners is probably in the tens of millions—I doubt if any other technical profession comes even close in terms of scale. (For example, according to the US Bureau of Labor Statistics, the US alone has approximately 1,330,000 "software engineers and programmers" [35].) It is likely that the majority of them are "classically" trained in current third-generation technologies and methods,[3] presenting an enormous inertial mass that is resistant to substantive technological shifts. As might be expected, this imbalance is also reflected in research investments, so that MBE-oriented research tends to be underfunded.

Therefore, to answer our question, we must look beyond just the technology. Specifically, we shall examine each of the following categories of factors:

1. Cultural and social factors
2. Economic factors
3. Technical factors

Unless we develop solutions that address all of the above, it is unlikely that we will see a greater penetration of model-based approaches in industrial software development.

But, before we delve into these different categories, let us first clarify what is the objective we are seeking.

2 The goal of MBE

> "If a builder has built a house for a man, and his work is not strong, and if the house he has built falls in and kills the householder, that builder shall be slain."
>
> *The Code of Hammurabi* (250), twenty-second century BC

> "An honourable work glorifies its master – if it stands up"
>
> Lorenz Lechler, *Instructions, 1516*

[2] In contrast, the same cycle in most engineering disciplines may take months or even years.

[3] Another source of problems is the exceedingly low entry barrier into the profession; basic programming skills can be picked up quickly and easily by amateurs. Consequently, many practitioners lack systematic and deep training in basics of computer science and engineering.

A parallel is sometimes drawn between the current state of the practice of software development of *complex*[4] software systems and the manner in which the intricate Gothic cathedrals were designed and built in medieval times [10]. Both are characterized by a vexing lack of hard evidence at the outset that the desired objective will be achieved (catastrophic failures were rather frequent in early cathedral construction), a high probability of undetected design flaws calling for major redesign in the course of construction (and, usually, at great expense), unpredictable and protracted completion dates, and cost overruns. Although there is much disagreement about the level of unreliability of software development (e.g., [11,13, 34]), many practitioners feel that the failure rates of software projects are substantially higher than in other more mature technical disciplines. But, as our society becomes increasingly more dependent on software and the cost of software errors grows (e.g., financial losses due to software errors, failures of safety–critical systems such as nuclear power station control systems), there is likely to be mounting pressure to improve on the current reliability of software in general.

The design and construction of medieval cathedrals involved skills that depended primarily on the long-term experience and knowledge of a small number of "master builders"—individuals who developed and honed their skills over decades of apprenticeship in many projects in which they were mentored by more senior and more experienced practitioners of their craft [10]. Their mastery was often based on intuitive understanding and standard "rules of thumb" (i.e., what we refer to sometimes as "design patterns"), which emerged through lengthy experience and whose validity and trustworthiness in a particular application were gauged mostly on the basis of the reputation of the master rather than on demonstrable analytical argument. (As with the current well-known "feature interaction" problems encountered in various types of software, one difficulty with this approach was that it was hard to predict when such intuitive rules might be at odds when combined with each other). Consequently, it took decades and even centuries to complete the construction of these edifices, with many false starts and failures, and, despite their undeniable beauty and extreme levels of craftsmanship, with clearly discernible and sometimes even jarring construction defects that would cost modern-day architects their professional licenses.

In contrast, most modern engineering efforts are run-of-the-mill affairs, based on past successes and a solid understanding of the underlying physical principles involved and knowledge of suitable predictive methods. Fundamental design failures are so rare that, when they occur, they are typically front-page news. In general, there is high confidence that a proposed design will end up being implemented close to its intent. Even when something quite innovative and experimental is attempted, such as, for example, Burj Dubai, the world's tallest building [6], there is still a very high degree of confidence before construction begins that the design is both feasible and adequate.

The key characteristic that we are seeking here is *reliability*. This has two fundamental components:

(1) *Reliability of designs*, that is, confidence that a proposed solution will fulfill its requirements and do so more or less within agreed-on cost, resource, and time constraints.[5] This implies predictability, that is, the ability to accurately assess the feasibility and cost associated with a design *before major investments are made in its realization*.

(2) *Reliable transmission of design intent* from specification to the implementation. If a design has been certified as valid and sufficient through various analyses, then it is crucial that the corresponding realization is an accurate rendering of that design and not otherwise.

Conventional engineering disciplines achieve design reliability through extensive use of science and mathematics. These powerful tools are at the core of the high degree of success of modern engineering. Unfortunately, applying them to software has met with very limited success, partly because software is much less constrained by physics than any other engineering medium. The laws underlying today's software are much less predictable and, consequently, much less amenable to formal mathematical treatment. Part of the problem lies in the highly non-linear and almost chaotic nature of most current programming languages, in which a minute difficult-to-detect flaw in logic can easily lead to seemingly disproportionate consequences. (For instance, a single missing "break" statement in a program controlling a telecom tandem switch resulted in a prolonged outage of a large portion of the long-distance telephone network in the United States [22]). In contrast, most conventional engineering mathematics relies on approximating the inherently jagged and messy nature of physical reality using simplified mathematically comprehensible models that are, nonetheless, sufficiently close to that reality so that the difference is (usually) insignificant. Unfortunately, today's software technologies, being based purely on formal logic, are highly and inherently non

[4] In this discussion we are only interested in "complex" software systems, characterized by sophisticated functionality and numerous technical challenges and whose conception and construction require a variety of skills.

[5] Although, as demand for ever more sophisticated systems increases, we are hearing more and more about cost and schedule overruns of various large engineering projects. However, unlike software, very few of these actually fail. Most high-risk projects are terminated early in the development, before significant funds and resources have been spent. This is because it is possible to reliably predict the probability of failure sufficiently early in the project.

linear. Such systems are notoriously difficult to capture accurately through mathematical abstractions (which, after all, are idealizations of reality) often leading to terminal complexity.

The second key component of our objective, reliable transmission of design intent, can sometimes also be a major challenge in classical engineering. Problems mostly occur when the models used to specify and validate designs are not sufficiently accurate renderings of the reality they represent. For example, the actual materials used in construction may not fully match the assumptions used during formal stress calculations, or, manufacturability issues might lead to divergence from the original design intent. In addition, there are inherent process discontinuities involved that complicate and skew the transmission process: compared to design, construction typically involves different people with different skills using different tools. In addition, misinterpretations of design intent are possible due to incompleteness or lack of clarity in design specifications, and so on.

software, these issues are, at least in principle, less severe: the specification and the end product share the same medium—the computer. This greatly reduces and, in some cases, eliminates the problem of the process discontinuities, since a model can be evolved through a series of incremental refinements until, with the help of computer-based automation such as code generation technology [28], it eventually evolves into the system that it was modeling (e.g., [37]). Since the same tools, skills, and methods are used throughout (and, often, the same individuals) there is much less likelihood of design corruption. Moreover, with its unparalleled flexibility, the computer, which is the shared medium of both the specification and the end product, can be used to automate much of the menial work involved in the design transfer (e.g., code generation using model transformers and compilers).

In summary, our goal is to evolve software development into something that is at least similar in its methodology to traditional engineering, while, at the same time, taking advantage of the unique nature of software. To help us identify what this implies, we cite the key factors identified by Robert Baber that distinguish craft from engineering [2]:

1. *The existence of a substantial body of applicable scientific theoretical knowledge.* As a rule, theory simplifies design by reducing solutions to their fundamentals and clearly explaining, how, why, and when a particular solution works. Although some core theoretical principles underlying software development have emerged over time, with the exception of a few specialized domains (e.g., compiler technology, relational database theory), this knowledge is far from being comprehensive, systematically organized, or even agreed on by the community.
2. *Systematic and thorough training of practitioners* in the established knowledge of their field. The rapid and rather chaotic development of software technologies and the low-entry barrier for acquiring basic programming skills have resulted in a profusion of inadequately trained practitioners, who are often not even aware of the major gaps in their knowledge.
3. *The systematic and disciplined application of such knowledge in practice.* Despite paying lip service to reuse, many software practitioners like to think of software development as comprising a process of continuous invention, with each new task requiring an original solution. The end result is that there is much needless and inefficient reinvention in software development. Instead of invention, what is truly needed is inventiveness, that is, the creative selection and application of proven solutions.
4. *A sense of individual "professional" responsibility towards the clients and end users.* This entails a thorough understanding and appreciation of the business needs of a product and a sense of obligation to fulfill those to the best of one's ability. Among other things, this means that, wherever possible, the definition of a product's requirements is a shared responsibility of both clients and designers.

Notice that only the first of these deals with technological aspects, while the others all relate to non-technical factors. With that in mind, it is clear that, as a technology, MBE can have an impact only on the technical elements. Therefore, we must seek for solutions in other domains as well.

3 Cultural and social factors

> "The greatest obstacle to discovery is not ignorance – it is the illusion of knowledge".
>
> – Daniel J. Boorstein, American author

> "He that knoweth not what he ought to know is a brute among men".
>
> – Aristotle, sixth century BC

Stephen Mellor, one of the pioneers of model-based methods, likes to tell the following story: When asked to predict when modeling will become mainstream practice in industrial software development, his response was "in three years time". Unfortunately, he goes onto say, he has been giving this same answer to that question for the past 20 years.

Although there do not seem to be any verifiable statistics about the degree of penetration of MBE in industrial practice, as the above anecdote illustrates and what most MBE proponents know from experience, it is still very far from being pervasive.[6] To a technically minded individual, particularly

[6] A rough estimate, based primarily on the author's several decades of personal experience, is that no more than 15 % of all industrial software projects currently use modeling substantively in their development.

to one who may have witnessed first hand the major advantages of MBE, this slow pace of adoption may be puzzling; especially, as noted earlier, given that there is sufficient credible publically available evidence confirming those advantages (e.g., [15,20,37]). Clearly, this is proving insufficient to settle the argument, implying that there are other factors at play that need to be understood.

One of the primary influences here is the set of behaviors and beliefs of its practitioners; i.e., the "culture" of software developers. Therefore, which particular culture-related factors are the primary impediments to greater and faster penetration of MBE in practice? Briefly, they can be summarized as follows:

1. *Inadequate or flawed understanding*—Given the relentless pressure of modern industrial software development, it is not surprising that many developers and development managers who could benefit from applying MBE are simply not sufficiently informed. Many rely on second- or third-hand opinions and are under the impression that modeling consists mostly of using expensive computer-based tools to draw unproductive "pretty" (UML?) diagrams, which have, at best, a very remote connection to the actual implementation. Consequently, they perceive it in the same way as they view documentation; that is, something to which one may have to pay lip service for reasons of political correctness, but which, in the final analysis, is a distraction and a waste of time that squanders away precious time and deflects resources from the only task that matters: programming. This may seem like an issue that is easily resolved through education. Unfortunately, this is where it combines with another culture-related factor: the technology-centric mindset of many software developers (see below). It is not that developers are not interested in learning. On the contrary, most are eager to keep up with what they see as the "latest and greatest" methods and tools by talking with their peers, by reading technical references (books, manuals, articles), by following blogs and web postings of recognized industry "gurus", etc. Unfortunately, this type of professional improvement is often limited in scope to established mainstream computing technologies, thereby precluding any radical shifts from the baseline. For example, while C++ programmers may be induced to try and even switch to more recent languages such as Java or C#, which are conceptually close to their expertise, it is unlikely that they will just as readily switch to a modeling language.
2. *Technology-centric mindset*—As mentioned earlier, programming is a highly seductive experience. No other technological medium offers such power at so little physical effort and cost. This naturally attracts creative individuals because, in a sense, it amplifies their creativity. Unfortunately, it also fosters a mindset which is quite often more focused on the existential pleasure of programming than the end product itself. Again, this impedes any kind of significant technological shift. Practitioners invest significant time in perfecting their skills related to a particular technology and are, thus, disinclined to switch—even if the newer technology may be better suited to the problem at hand.
3. *Lack of system perspective*—In my consulting work, I have noted that many of the programmers working on complex systems lack an understanding of the system as a whole. Moreover, except in a very superficial sense, many of them are comfortable with this situation and show no strong interest in expanding their system knowledge and even less in learning about the business value that the system brings to its owners and users. There is even a strong cultural element that hints at a certain level of disdain for end users because they may not fully appreciate the "elegance" of the technology. Part of this can be ascribed to the technology-centric mindset cited above. However, in this case, there is an additional related factor at work: *a lack of abstraction skills*. It is perhaps surprising that in software, which is largely based on manipulation of non-physical (i.e., abstract) entities, it is difficult to find practitioners with sufficiently developed abstraction skills to act as system architects; that is, individuals who see beyond the technology.[7] While some people are better at abstraction than others, it is my firm belief, borne out in part by experience, that basic abstraction skills can be taught. Unfortunately, given the technology-centric mindset of many practitioners, they simply fail to see the need for honing such skills, since they do not feel responsible for the system as a whole.

The above factors all contribute to the slow propagation of advanced technological shifts such as MBE.[8] So, what can we do to circumvent or at least mitigate this hurdle?

In my opinion, the simple and by far best answer is: education. Although nothing can replace experience—and designing the architecture of a complex software system should generally be assigned to experienced professionals—it is, nonetheless, possible to instil some of these higher-level skills (or, at least, an awareness of them) in graduates of

[7] Note that abstraction should not be interpreted as ignoring technological factors in software design, particularly since technological constraints can have a fundamental impact sometimes on the architecture of software. However, it does mean that only the essential aspects of the technology need be considered and that the application rather than the implementation technology should be the foremost concern of the implementers.

[8] A more careful analysis of the core technical skills required for model-based software development reveals that they are, in essence, no different than the core skills required for traditional programming—except that they are based on a different technological foundation. These core skills are, in fact, technology independent.

software engineering curricula. These include at least the following advanced topics, which I find under-represented in most software engineering courses I have seen to date:

- *A fundamental understanding and appreciation of system design*; we need not only more system architects but also more programmers who are capable of understanding the systems they are developing
- *A basic understanding of human factors* related to software systems (more on this below)
- *Knowledge of basic business economics* (if nothing else, this will make them understand why a technologically superior solution may not always be the best one for a particular problem).

I believe that, instilling this type of knowledge in software engineering graduates will result in a new, more open-minded generation of software professionals capable of selecting and exploiting whatever technology is best suited for the task on hand, whether it be MBE or assembly language. Sadly, I do not hold out much hope that these skills can be retrofitted to the existing large body of "classically trained" software professionals—it does not seem feasible.

To help stimulate interest in model-based engineering, it would be extremely useful to establish a generally accessible repository of information that documents industrial experiences with model-based methods, as well as to conduct systematic studies of such data. One promising effort in this vein is the Empirical Assessment of MDE (EA-MDE) project being conducted at the University of Lancaster [14,15].

It is unrealistic to expect the culture involving such a large population to change overnight; it will, undoubtedly, be a long and protracted process spreading over many years. One concern is that the highly volatile software industry—susceptible as it is to claims of silver-bullet solutions—may not provide the stability necessary to realize such a goal. However, without such a major cultural shift, I cannot see any hope for greater adoption of advanced technologies such as MBE or its successors. We have been mired in the current (third) generation of programming technologies for far too long.

4 Economic factors

"A bird in the hand is worth two in the bush"

– Old English proverb

"You miss 100% of the shots you don't take"

– Wayne Gretzky,
Canadian hockey player

Despite the almost inertia-less nature of software, industrial software development is often a heavyweight process that involves significant investment in training and tools. Introducing a new methodology requires both time and financial resources. Of the two, it is time that is often the more problematic, since developers are obviously less productive during the time taken up by training. In today's extremely competitive climate in which corporate performance is evaluated on a quarterly basis, it is difficult for management to sacrifice likely short-term income in favor of vague future gain. Furthermore, even after staff are trained and the requisite tools purchased, it is almost certain that productivity will drop until the development teams learn how to best exploit the new technologies. Thus, the hit on productivity can be protracted and can even lead to damaging loss of market share.

On the flip side, corporate management must take into consideration the possible "exhaustion point" of their current development technologies and processes, that is, the point at which these will become impediments to productivity and, ultimately, market share. For example, these days we are seeing dramatic changes in the world of mobile communications and computing, in which long-established market leaders relying on established technological platforms are finding their share of the market dropping precipitously, forcing them into catch-up mode [4]. Therefore, the ability to respond quickly to market pressures with new high-quality software is crucial to maintaining a competitive advantage. Since MBE has demonstrated its potential for dramatic improvements in productivity and quality compared to traditional software development approaches it should be of great interest to product managers. Published examples of corporate-level strategies for introducing MBE into legacy environments can be found and should be studied carefully [17,30]. Again, establishing a publically available database of information related to industrial experience with MBE would be a very useful facility to this end.

When it comes to tool purchases, one often encounters a somewhat paradoxical situation in industry: despite the fact that computer-based tools are the basic means of production of software developers and, consequently, have a major impact on their productivity, it is often the case that it is difficult to get approval for purchasing new tools. As far as MBE tools are concerned, their cost ranges from free to several tens of thousands of dollars (for certain highly specialized tools). Most of the core commercial tools, such as model authoring tools, cost less than $3,000. However, it is difficult to make a convincing case for such purchases, since the expected productivity gain that such a tool might bring is difficult to predict with any level of certainty.

Of course, open source tools are usually free, but this sometimes turns out to be a disadvantage. This is because, at present, many open source tools are not sufficiently robust or functionally capable to cope with the technical challenges of

complex industrial projects. However, their availability has created expectations that development tools should be either inexpensive or free. This trend is driving the price of commercial tools downward reducing vendors' profit margins to the point where their economic viability is in jeopardy.[9] A common response by vendors to this is to produce general-purpose tools that cover the broadest possible market space, which, unfortunately, clashes with the industry need for ever more specialized tools and the MBE trend towards domain-specific languages.

Fortunately, it seems that a solution to the tools conundrum is starting to emerge, but it is not without cost to industrial users of MBE. It is based on industrial funding of open source tools development. This is primarily being driven by corporations who have concerns about the long-term support for the tools they are using to develop their software as well as being locked into a single vendor (who may not survive for the full lifecycle of the product). For instance, in the aviation industry, the lifespan of a product may exceed 50 years, which surpasses by far what commercial vendors are willing to commit. As a result, such users are turning more and more to open source tooling that will free them from a vendor lock-in situation.

Exemplary representatives of this type of initiative can be found in the work of the Eclipse Foundation [8], which has helped establish a number of consortia and industry working groups whose primary purpose is to share the cost of developing industrial strength open source tools that are of core interest in their business. Another positive consequence of this is a much stronger interest in defining associated industry standards.

Still, it is hard to obtain corporate approval for such investments for the same reasons that it is hard to get approval for purchasing tools. In fact, the initial corporate outlay required to invest in open source development often exceeds the amounts needed for purchasing commercial tools, making it an even harder case to argue—not to mention the complex intellectual property issues that need to be resolved in such cases. Clearly, the more enterprises are involved the lower the individual cost to each of them, and, therefore, the key lies in finding a critical mass of partners with similar tool requirements willing to participate in such consortia. This, unfortunately, is highly susceptible to the cultural issues discussed earlier.

[9] Contrary to popular impressions, developing software tools requires major research and development investments, since a professional grade tool must not only have the required functionality but it must also be robust, scalable, and highly usable—all of which requires significant effort to achieve. Furthermore, it also requires documentation, training materials and courses, and a support and maintenance team.

5 Technical issues

> "If we have learned anything from the history of invention and discovery, it is that, in the long run – and often in the short one – the most daring prophecies seem laughably conservative".
>
> – Arthur Clarke, *The Exploration of Space*

> "It would appear that we have reached the limits of what is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years".
>
> – John von Neumann,
> computer pioneer (cca. 1949)

We finally come to the part that is likely of greatest interest to readers of this journal: the technical advancements that are needed to solidify MBE into a mature and reliable technical discipline. Most MBE technologies are still in early stages of development, although some of the core capabilities have been used successfully in industry for at least the past two decades. However, with increased exposure to industrial problems, the current shortcomings of these early technologies are becoming more evident, demonstrating quite clearly that more research is needed. These limitations are both pragmatic and theoretical. Pragmatic deficiencies relate to the utilitarian aspects of the technologies; that is, engineering issues such as scalability, usability, and reliability. Theoretical issues, on the other hand, have to do with the lack of a deeper understanding of the essence behind technology that is needed to determine when and why some things work and when and why they do not.

At present, my personal impression is that, in terms of innovation, we are much further ahead on the pragmatic side than on the theoretical one. I suspect that this is due to the severe competitive pressures to which commercial enterprises are typically exposed, often forcing them to devise creative and original solutions to deal with immediate customer problems they are facing. However, these are typically very specific solutions to very specific cases and, because they lack a systematic theoretical foundation, we often have difficulties understanding how, why, and when they might be applicable to other cases. Like those medieval master masons, we need to take the next qualitative step that would take us from craft to engineering, one that is based on firm theoretical underpinnings. It is my belief that there is already sufficient empirical knowledge available so that we can start the development of such a theory. This holds for each of the areas of technical development discussed below.

In a seminal article in this journal, Robert France and Bernhard Rumpe outlined a comprehensive research roadmap for model-based engineering [12]. From an industrial perspective,

however, some of these areas are of greater priority; in what follows we attempt to identify those technical developments that, in my view, *must* take place before industry can *confidently* rely on MBE. These can be grouped into three major categories of technical challenges:

1. *Capability challenges*: necessary functional capabilities that are either inadequately developed or missing altogether
2. *Scalability challenges*: various capabilities required to deal with large and complex system models
3. *Usability challenges*: the ability to reduce or eliminate the accidental complexity that occurs in the development of large systems to fit what can be managed by normal human intellectual capacities.

Some readers may feel that challenges related to development process are equally, if not even more crucial to the emergence of a successful software development discipline. While I fully concur that these are indeed issues of the very highest priority, I feel that they are universal and more or less independent of the particular computing technology being used for development, model-based or otherwise. The only potential distinction in case of model-based approaches in this regard might be their greater reliance on computer automation.

5.1 Capability challenges

MBE technology has moved forward significantly in the past half decade. There are even industry-wide standards for complex model transformations, such as OMG's MOF QVT [24] and model-to-text technology recommendations. Nevertheless, we are still far behind from what is needed to meet our goal. In particular, we need major advances in both the theoretical and pragmatic aspects of the following technical capabilities:

1. Modeling language design and specification
2. Model synchronization
3. Model transformations
4. Model validation.

Although each of these areas is discussed individually below, as we shall see, they are quite intertwined.

5.1.1 Modeling language design and specification

The experience with UML, which is almost certainly the most widely known and used but also the most criticized modeling language, has demonstrated that we have much to learn yet on how to properly design such languages. Solving this problem is particularly urgent given the increasing need for domain-specific modeling languages.[10] In contrast, the design and specification of classical programming languages is as close to a systematic technical discipline as we have in computer science. The theory behind the design of such languages is both sound and well developed. Unfortunately, there are some important differences between programming languages and modeling languages that preclude a simple transfer of methods developed for the former to the latter.

Language syntax Perhaps the most striking difference stems from the fact that modeling languages, in addition to serving as technology specification language (i.e., blueprints for construction of programs), must also be designed to support the crucial human communications function of models.[11] (This feature of engineering models is often neglected, but, it is one of the primary benefits of modeling, on par with the ability to use models for predicting system characteristics.) It implies a much greater emphasis on *ensuring that the concepts and the syntax of a language are interpreted easily and correctly by human observers*. This, however, requires more than traditional technical skills. For example, it means deciding on what are the most direct and most intuitive ways of expressing a concept, which clearly requires a deep understanding of the domain and the psychology and culture of its practitioners. Should a graphical or textual syntax be used, and, if the former, what is an appropriate graphical shape or form? Note that a model will likely need to be examined by different categories of stakeholders, who will want to view it from their specific viewpoints, based on their specific ontologies and concerns. Therefore, it must be possible to construct multiple different concrete syntaxes for a given language, including the ability to create new ones on an as-needed basis.

Of course, devising the concrete syntax of a language is an exercise in human–computer interface design, which is known as a difficult design problem demanding sophisticated and diverse skills [7]. Unfortunately, the issue of designing the syntax of computer languages, particularly those based on graphical forms, has not received due consideration,[12] with the notable exception of the recent work reported in [21]. This is clearly an important area of research.

[10] Recall that one of the keystones of model-based engineering is to move the task of software design closer to the problem domain and away from the computing technology domain. Consequently, the push towards domain-specific languages is a natural outcome.

[11] In contrast, the primary criterion in the design of programming languages is that they must be unambiguously interpreted by a compiler (such as the rigid syntactic rules that ensure computer-based parsing), a requirement that invariably takes priority over any concerns about readability.

[12] The example of the much reviled and rather cumbersome COBOL syntax, designed naively with a view towards making programs readable by non-programmers, demonstrates the difficulty of getting this aspect right.

Abstract and incomplete models Another important difference that separates modeling languages from programming languages is that not all models need be complete; often a rough, incomplete, "sketch" is sufficient. Yet, we still want to get maximum benefits from such sketches, implying the possibility of formal analyses despite their incomplete nature. Programming languages, on the other hand, which are primarily targeted at computers (i.e., compilers) are constructed to be utterly unforgiving of even the slightest omission. For instance, most compilers refuse to produce a line of output unless all the rules of syntax are fully satisfied. This often requires significant effort and painstaking attention to detail that may be counterproductive in early stages of design when high-level choices are being made. *The design challenge then is to design a modeling language in such a way that a model can still be semantically meaningful even when it is incomplete*. For instance, a state machine specification in a very early design model may completely omit some transition triggers and other details, but it should still be possible to execute such a model even though it is in a syntactically incomplete state. This ability is particularly important for modern agile development, in which it is crucial to be able to create models of candidate designs with as little effort as possible and to evaluate them as early as possible. (One of the known problems of prototypes constructed using traditional programming languages is that the unrelentingly pedantic nature of compilers requires substantive detailed effort be put into a prototype, which greatly slows the rate of design iteration and also leads to a reluctance to discard inadequate candidate designs.)

Modeling language semantics Another major issue with language design is the definition of semantics. It should be possible to define the semantics of a language such that:

(1) the meaning of its constructs are precise enough to avoid misinterpretation by tool implementers
(2) it is internally consistent and
(3) it is amenable to various types of formal analyses.

All of the above point to some kind of formal mathematically based approach to semantics specifications. But, it is still very much an open question as to which formalisms, if any, are most suitable for such a purpose. A general formalism, capable of covering a broad range of different languages is necessarily based on a lowest common denominator, such as first-order logic[13] (e.g., Common Logic [16]). The problem with using low-level formalisms like this is scalability: expressing the complex semantics of the types of phenomena encountered in various domain-specific languages requires extremely complicated specifications that are not only difficult to produce but, what is even more worrying, are also very difficult to validate. Higher-level formalisms, on the other hand (e.g., Petri nets, finite state machines), while more expressive are less widely applicable. This can be a problem when it comes to dealing with situations where multiple different modeling languages are used to specify a complex system, since there may be semantic incompatibilities between the languages. The answer seems to lie in some type of layered combination of formal languages, with a shared low-level semantics foundation.

Without a formal foundation of this type for modeling languages it is difficult to see how we can ever reach our objective of transforming software development into a reliable and repeatable technical discipline. Therefore, this represents a truly crucial research area that must receive more attention than has been the case to date.

5.1.2 Model synchronization

Modern technical systems often involve multiple different subsystems all of which will need to be modeled if a model-based approach is used. The software of a modern automobile, for example, will have specialized software to control the engine and fuel injection system, software for the braking system, software for the internal entertainment system, and so on. Given that the software controlling these various elements may be based on different paradigms (e.g., event-driven versus time-driven, hard versus soft real time, safety critical versus non-critical, etc.), their models may be specified using different domain-specific modeling languages and may be written by different groups (even different companies). Moreover, these models coexist with models of other non-software based parts of the system as well as to the requirements. Since the various components of a complex system are usually interconnected, it is very helpful if the corresponding models are similarly linked. This provides the ability to understand and reason about the system as a whole, including the important ability to predict the ramifications that a requirement or design change in one part of the system might have on other parts of the system.

In general, a specification of a complex system will include many models specified using multiple different languages. In addition to specialized models of different parts of the system, there may also be models at different levels of abstraction and models representing the system from different viewpoints from different stakeholders. Similarly, there may also be documents that explain the

[13] In fact, mathematical logic on its own is probably not sufficient for specifying language semantics in domains where quantitative aspects of physical phenomena, such as the duration or resource limitations, play a key role.

design rationale and workings of the system and its parts as well as the program code that is a consequence of the models. All of these different artifacts need to be synchronized to ensure that the information is consistent. As systems grow in complexity, this task quickly becomes onerous and even unmanageable without the help of computer automation. A practical solution to this problem is one of the most sought after benefits that industry expects of model-based engineering.

The simplest solution to this issue is to establish crude computer-based pointers ("hyperlinks") between coupled elements, even if they are in different models. However, this minimal approach does not ensure proper synchronization in which a change in one element results in a corresponding consistency-preserving change in its related elements. To achieve that objective, the links between the interdependent model elements have to be "semantics aware", meaning that they preserve the desired semantic relationships between the linked elements. Antkiewicz and Czarnecki [1] provide a comprehensive review of this field, identifying the various forms of "synchronizers" that may be needed and ways of realizing them.

Despite significant academic research in this area and some industrial advances, we still do not have a ready-made industrial-strength solution. There are various technological hurdles that stand in the way, such as inadequate or missing interchange standards between different modeling tools, lack of suitable APIs, etc. However, even if those lesser technical issues are resolved eventually, there still remains the problem of scale. Namely, in many industrial systems, the sheer volume of such interconnections (and their transitive closure) can be overwhelming. This not only requires significant computing and storage resources, but more critically, it involves significant overhead. That is, during development, such linkages require extra time to set up— and time is almost always in short supply in modern industrial practice. (It is rare for project managers and developers to get rewarded for making future projects successful.) In a very concrete sense, they are like technical documentation, a putative investment in the future, and, consequently, are the first things to be dropped when deadlines loom. Moreover, once such (possibly complex) synchronization links have been set up, they become an inertial impediment to design change. After investing substantial time and effort in setting them up, developers may be more inclined to patch up a bad design rather than do a proper redesign and be forced to go through the linking effort all over again.

An industrial-strength model synchronization facility has to be very lightweight yet effective. The key seems to lie in some form of hierarchical componentization of designs, such that most changes are localized and do not propagate beyond their component compartments. This type of isolation is particularly difficult to achieve in software, partly because even independently designed software components are often coupled implicitly and insidiously by virtue of sharing (and contending for) common computing resources.

5.1.3 Model transformations

Model transformations are used when it is necessary to generate programs, documentation, or other derived viewpoint-specific models from a given input model. It involves the generation of new entities or updating of existing ones from models, based on a formal automatable or semi-automatable process. Note that model synchronization may (and typically does) involve model transformations.

What is missing in this particular area is an agreed-on systematic theory of model transformation design. Fortunately, with developments, such as OMG's QVT specification [24], and the work of numerous researchers [25–29], we are seeing the emergence of such a theoretical underpinning. In fact, this may be the one area of MBE where we have progressed furthest in terms of theory. Still, we are far from making the writing of model transformations an established and repeatable technical task. This may be surprising at first, since it would seem that model transformation is simply a generalization of the well-known problem of program compilation. But, as pointed out in Sect. 5.1.1, there are some major differences between modeling languages and programming languages that require innovation.

Nevertheless, there seems to be much that could be reused from compiler theory and applied to model transformation problems—but this has not happened in any substantive way. In the time-honored tradition of software engineering, rather than standing on the shoulders of giants, most of what we know about model transformation has been (re-)invented from scratch. For example, the most common method for code generation is to translate the model into a standard third-generation programming language and then use the compiler for that language to get to the final executable. This has a number of major drawbacks. First, it creates a discontinuity between the executable code and the source model, which greatly complicates model debugging. Second, because the programming language compiler has no understanding of the semantics of the original modeling language, many opportunities for program optimization are lost in the process.[14] Finally, the presence of a programming language version of the model tempts many developers who are more familiar with programming than modeling into making "manual" modifications to the generated program, which can be then be easily lost during subsequent changes to the original source model. Interestingly enough, most of these people would be

[14] In fact, this two-step approach is analogous to the old CFront solution used in early C++ compilers—an approach long abandoned because it resulted in inefficient code.

horrified if someone asked them to do the same for compiler generated code. (To be fair, one of the primary reasons why this two-step approach is used is precisely because many developers demand to see the generated code, mostly because they do not have sufficient confidence in the code generator.)

Due to the lack of theoretical underpinnings, the design of code transforms in industry is still a craft, typically left to a small number of in-house experts, who, like our medieval master masons, often prefer to keep their skills to themselves. This, of course, exacerbates the problem of lack of trust in the outputs of model transforms.

There is no obvious reason why a systematic and reliable theory of constructing model transforms cannot be developed and why many of the principles of compiler construction cannot be used to bootstrap that theory. The fact that each domain-specific language requires its own set of model transforms (code generators, viewpoint generators, document generators, etc.) makes this an even more pressing need.

5.1.4 Model validation

The term "validation" is used here in a broader sense and covers all processes used to determine whether a design and its corresponding implementation satisfy system requirements. It includes both static and dynamic methods. Static methods include formal automated or semi-automated analyses of designs, such as checking the safety and liveness properties of designs, whereas dynamic methods include simulation and testing.

A known problem with traditional software development, and particularly in case of complex systems, is that there is a great deal of uncertainty about the validity of the chosen design until enough of it has been implemented and assembled for meaningful system testing to commence. Consequently, serious design flaws are sometimes discovered very late in the process (often not until the integration phase), when it is generally very expensive to fix them. Much of this is due to the informal nature of design specifications as well as the technical difficulties of any type of formal analyses of programs.

Clearly, the sooner a candidate design if validated, the better. However, the trustworthiness of such validation is fundamental. Without a formal foundation for modeling languages it is very difficult to make reliable predictions about the validity of a design—hence, the importance of formality noted earlier. For instance, by basing modeling language on well-known formalisms, such as Petri nets or Milner's Calculus of Communicating Systems or its derivatives [19], which were designed to facilitate formal validation, we may be able to finally break through the current barriers to the practical application of formal methods to industrial problems. So-called *synchronous* languages, such as Esterel [3], are perhaps the best and most encouraging examples of the practical viability of such an approach.

Static validation techniques These are, invariably, based on formal computer-based analyses. In addition to the classical formal methods for assessing the logical correctness of a design (i.e., safety and liveness), such as model checking and theorem proving, it is also crucial for us to develop practical methods for predicting the *qualities of service* of a proposed design—characteristics such as performance, delays, response times, availability, reliability, safety, etc. In general, the problem of assessing the *quantitative* properties of a software system has not received sufficient treatment to date. This is partly due to the fundamentally flawed but widely held assumption that software correctness reduces to logical correctness and that, in general, software and physics should not be mixed. However, with the increasing use of the internet for commerce and other social activities, the importance of this facet of complex software systems is finally being recognized. (A logically correct program that takes 25 h to compute tomorrow's weather is of little value.) Critical advances in this direction have already been made in certain limited areas, such as schedulability [33] and performance [31], but we are still in great need of a much broader set of capabilities of this type for at least the most critical system properties, such as security and safety. Until such tools become readily available and are sufficiently dependable, we really cannot claim to have a mature technical discipline.

Dynamic techniques Dynamic techniques involve testing designs by executing models. Since exhaustive testing is generally not feasible and, given that both time and resources dedicated to testing are limited, a primary technical challenge here is to determine the set of test sequences that best serve the validation purpose. As with many other aspects of model-based approaches, this is still very much a research problem. It is part of the general problem of software testing, although testing of models presents some unique challenges given the more abstract and more domain-oriented semantics of the languages used. In particular, it would be useful to investigate how knowledge of language semantics can be used to maximize test coverage.

Dynamic validation techniques require executable models. Note that not all models or modeling languages need be executable, and there are definitely cases where it is not even helpful. For instance, predicting the integrity level of a safety–critical system is best done analytically using a mathematical model. In that case, executability adds no value. However, in many situations the ability to execute a model early in development provides a number of important advantages. First, it requires a formal and consistent modeling language—characteristics that are generally beneficial as explained earlier. In addition, we must not underestimate the

psychological advantages that come with having an executing model, even if it is a very abstract one. As noted before, the ability to rapidly move from idea to reality is a prime motivating factor for many software developers (it is one of the main reasons that practitioners generally prefer agile approaches to development). Seeing one's design ideas "in action", no matter how trivial, provides positive mental re-enforcement, acts as a clearly identifiable progress marker, and, perhaps most importantly, it helps designers develop an intuitive understanding of the problem that would be difficult to attain by purely conceptual analysis.

Model executability for validation purposes requires a lot more than just simple code generation. It is usually better to base it on interpretation, that is, through a language-specific virtual machine, because that is a way of circumventing several practical problems associated with direct code generation. For one, it more readily supports execution of incomplete models, such as the abstract and incomplete models generated in the early phases of design. This capability is critical for early validation of designs. Naturally, a language-specific model execution system (virtual machine) will likely require some manual or other type of intervention when it runs into a situation where the detail necessary to continue execution is absent. But, implementing such a capability is not a particularly difficult technical challenge. Another advantage of this approach is that it better supports *model-level monitoring and debugging* of the model. Experience has shown that it is extremely difficult to debug a model if the record of its execution is expressed in terms of the generated code rather than the original source model.

Although implementing such a model execution system does not present any insurmountable technical challenges, they are not readily available to industrial users. A number of commercial model simulation systems for modeling languages such as UML, SDL, or AADL, are available, but none of them combine all the characteristics described above.

5.2 Scalability challenges

I have already described the scalability problems associated with formal methods and with model synchronization. These are just part of the overall problem of dealing with large and complex industrial software systems. Some of the issues related to scale are purely due to inadequate tool implementations. However, a few of them are still open research problems. Two of the most critical in the latter category are the issue of incremental model change and the issue of model fragmentation. Both of these have become critical and are presenting major hurdles to industrial users of model-based technologies.

Incremental model change problems occur when a change is made to a model that affects other models linked to it, which, in turn, requires model synchronization. When the models in question are very large, unless special care is taken, this can be expensive and time consuming, creating a major development bottleneck. For example, when a change is made to a model, only the affected parts of the corresponding generated program code should be modified and recompiled as opposed to regenerating the code for the entire system (this can take many hours!). This has proven to be one of the most serious practical problems associated with MBE methods that depend on model transformations such as code generation. Unfortunately, determining the scope of the target model changes required for a given source model change is a difficult task, and much dependent on the languages used, the nature of the system, and even the tools available. In other words, it is still in the realm of craft rather than engineering.

Many of the current generation of model authoring tools assume that the entire model is directly and efficiently accessible in fast main memory of the computer, to facilitate the execution of frequent operations such as search. However, for larger complex systems, this is not always practical. Part of it has to do with sheer size: the models often exceed the storage capacity of an average developer workstation, resulting in frustrating and, sometimes, crippling memory thrashing problems. Recall that in most cases models are stored in the form of interconnected graphs, so that everything is, ultimately, connected to everything else (not to mention the case of multi-models). Consequently, basic things, such as "affected by" searches or code generation take an inordinate time to complete.[15] This too has proven to be one of the principal impediments to effective exploitation of model-based methods in industry.

Once again, the key technical advance that is desperately needed to cope with this issue is a proper theoretical framework accompanied by corresponding industry standards for partitioning models into independently packaged units and ways of manipulating them efficiently.

5.3 Usability challenges

Last, but *certainly* not least, there is the core issue of usability. This is a tools implementation issue, but, just because it is, we should not dismiss it lightly. In fact, it is a substantive and important research problem: we do not currently have clear guidelines on how to design our tools in a way that minimizes the accidental complexity they introduce into development. In my consulting experience, I have seen a wide variety of model-based engineering tools used in industrial practice,

[15] Of course, similar scalability issues occur in traditional programming language systems as well. However, because these programs are primarily textual rather than based on semantic linkages, these types of sophisticated searches are rarely even considered.

and I can ascertain that, overcoming the shortcomings of tools is perhaps the single most difficult hurdle to the effective exploitation of model-based methods. Specifically: the tools are far too complicated for most developers.[16]

Software development tools are designed by software developers. Unfortunately, developers far too often lack a deeper insight into their users needs. Often, they imagine someone not unlike themselves, someone with a deep understanding and appreciation of computing technology. But, given that many (most?) modeling tool developers are using traditional third-generation programming technology, this image is likely to be dangerously inaccurate.[17] Worse yet, very few of them are familiar with the basic precepts of user-oriented design and, what is even more worrisome, unaware of this. Hence, usability is often viewed as a second-order graphical user interface problem; to be addressed *after* the architecture of the tool has been decided. But, usability is often a deep architectural issue that cannot be easily retrofitted on a framework not designed to support it.

Designing an effective tool is similar to the problem of designing a modeling language in that it requires a thorough understanding of the problem domain, the culture of its practitioners (i.e., tool users), the theory of modeling language design, and usability. Sadly, few tool developers and tool architects are adequately equipped for this. Therefore, it is essential to involve problem domain experts as directly as possible in the processes of designing both languages and tools. (Moreover, it is important to pay attention to and respect the contributions of such individuals and teams—like many experts in other domains, software people sometimes tend to downplay the opinions of lay people.)

In addition, the education and training of software developers should be extended to deal with these issues. Qualified software developers must understand the importance of usability, and have basic competency in applying its core principles. (If nothing else, they must be sufficiently knowledgeable about usability to recognize their own limitations and when it is necessary for them to seek expert help).

6 Summary

Model-based approaches to software development and related technologies hold out the promise of significantly improving the productivity of software developers and the quality of the products they generate. This is because they are based on some of the same principles that have been proven effective in the past in software (raising the level of abstraction of software specification and raising the level of computer-supported automation), as well as in traditional engineering disciplines (use of models to reduce risk early and reliance on formal analysis methods). Industrial experience with this approach over the past decade and a half, has already demonstrated that is not just an empty promise.

However, we are far from being in a position to take maximal advantage of its potential. At present, it is still far from being a reliable weapon for taming software development; we do not really understand how to use it best, relying instead on the experience and knowledge of "master craftsmen". Unfortunately, there is not enough such experience to go around and even these experts cannot guarantee success, since the core principles behind their craft are still poorly understood.

I have attempted here to identify which key developments need to occur for model-based approaches to reach the status of a mature and reliable technical discipline. One of the main points is that just solving the technical challenges—which are daunting on their own—is not sufficient. A complete solution also requires dealing with cultural and economic issues. In fact, my sense is that these are probably the bigger challenges, since they involve factors that seem to be beyond our control as technologists. However, it is my firm belief that these can and will be overcome, mainly because market and social forces will reveal that the current technical approaches upon which this culture is based are no longer capable of providing adequate solutions to problems of modern software development. The key to progress lies in educating the coming generations of software practitioners to be more attuned to the needs of their users than their technology.

[16] An informal analysis of two of the most widely used commercial model authoring tools revealed that they both had in the order of 300 distinct menu items, many of them interdependent in ways that are not always obvious. Mastering such tools is, surely, a high challenge even for the most competent developers.

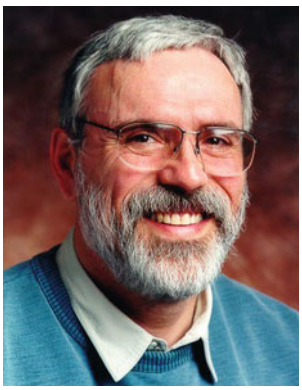
[17] I have no formal verifiable proof of this as a general conclusion, but I have been directly involved with the development of at least four major modeling tool projects (commercial and open-source), and it applied in all cases.

References

1. Antkiewicz, M., Czarnecki, K., et al.: Design space of heterogeneous synchronization. In: Lammel, R., Lammel, R., et al. (eds.) Generative and Transformational Techniques in Software Engineering II. Lecture Notes in Computer Science, vol. 5235, p. 46. Springer, Berlin (2008)
2. Baber, R.: Software Reflected. North Holland, Amsterdam (1982)
3. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
4. Bloomberg, Nokia's Share Slips as Unbranded Phonemakers, Apple gain Ground, http://www.bloomberg.com/news/2010-11-10/nokia-s-market-share-slips-below-30-as-smaller-vendors-grow-gartner-says.html, November (2010)
5. Bone, M., Cloutier, R.: The current state of model based system engineering: results from the OMG^{TM} SysML request for information 2009. In: Proceedings of the 8th Conference on Systems Engineering Research (CSER), March 2010. Hoboken, NJ (2010)
6. Burj Khalifa, http://en.wikipedia.org/wiki/Burj_Khalifa

7. Constantine, L.: What Do Users Want? Engineering Usability into Software. Constantine & Lockwood Ltd. (1995). http://www.foruse.com/articles/whatusers.pdf
8. The Eclipse Foundation. http://www.eclipse.org/
9. Einstein, A.: Science and happiness (Speech given at the California Institute of Technology). Science **73**(1893), 375–381 (1931)
10. Erlande-Brandenburg, A.: The Cathedral Builders of the Middle Ages (translated from the French by Rosemary Stonehewer). Thames and Hudson, London (2007)
11. Eveleens, J.L., Verhoef, C.: The Rise and Fall of the Chaos Report Figures. IEEE Software, pp. 30–36 (2010)
12. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Proceedings of 2007 IEEE Conference on the Future of Software Engineering (FOSE '07), pp. 37–54. IEEE Computer Society (2007)
13. Glass, R.: The Standish report: does it really describe a software crisis? Commun. ACM **49**(8), 15–16 (2006)
14. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceedings of 33rd International Conference on Software Engineering (ICSE'11), pp. 633–642. ACM, New York (2011)
15. Hutchinson, J., Whittle, J., Rouncefield, M., Kristofferson, S.: Empirical assessment of MDE in industry. In: Proceedings 33rd International Conference on Software Engineering (ICSE'11), pp. 471–480. ACM, New York (2011)
16. International Standards Organization: Information Technology—Common Logic (CL): a framework for a family of logic-based languages, ISO/IEC Standard 24707:2007 (2007). http://standards.iso.org/ittf/licence.html
17. Lin, C., et al.: Experiences deploying MBSE at NASA JPL. Frontiers in Model-Based Systems Engineering Workshop. Georgia Institute of Technology (2011). (www.pslm.gatech.edu/events/frontiers2011/2.1_Lin.pdf)
18. McLuhan, M.: Understanding Media: The Extensions of Man. MIT Press, Cambridge (1994)
19. Milner, R.: Communication and Concurrency, International Series in Computer Science. Prentice Hall, Upper Saddle River (1989)
20. Mohagheghi, P., Dehlen, V.: Where is the proof? A review of experiences from applying MDE in industry. In: Schiefendecker, I., Hartman, A. (eds.) Model Driven Architecture—Foundations and Applications (ECMDA-FA 2008), LNCS, vol. 5095, pp. 432–443. Springer, Berlin (2008)
21. Moody, D.: The 'Physics' of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng. **35**(6), 756–779 (2009)
22. Neumann, P.: Computer Related Risks. Addison Wesley, Boston (1995)
23. The Object Management Group: Compilation of SysML RFI—Final Report, OMG Document syseng/2009-06-01 (2009)
24. The Object Management Group: Meta-Object facility (MOF) 2.0 Query/View/Transformation (QVT), OMG document formal/2011-01-01 (2011)
25. Czarnecki, K., Helsen, P.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3) (2006)
26. Syriani, E., Vangheluwe, H.: De-/re-constructing model transformation languages. In: Electronic Communications of the EASST, 29: Graph Transformation and Visual Modeling Techniques (GT-VMT) (2010)
27. Varro, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: Proceedings of the 7th International Conference on the Unified Modeling Language (UML 2004), pp. 290–304. Lisbon, Portugal (2004)
28. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of Model Transformations in Practice Workshop. MTIP), MoDELS Conference, Montego Bay, Jamaica (2005)
29. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented metalanguages. In: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, pp. 264–278. Montego Bay, Jamaica (2005)
30. Oster, C.: Evolving Lockheed Martin's engineering practices through the creation of a model-centric digital tapestry. In: Frontiers in Model-Based Systems Engineering Workshop, Georgia Institute of Technology (2011). http://www.pslm.gatech.edu/events/frontiers2011/1.5_Oster.pdf
31. Petriu, D.: Software model-based performance analysis. In: Post-Proceedings of the MDD4DRES Summer School, Aussois 2009. Hermès Science Publications, Paris (2012) (in press)
32. Selic, B.: Personal reflections on automation. Program Cult Model-Based Softw Eng Autom Softw Eng J 14(3/4) (2008)
33. Sokolsky, O., Lee, I., Clarke, D.: Schedulability analysis of AADL models. In: Proceedings 20th International Parallel and Distributed Processing Symposium (IPDPS 2006). IEEE (2006)
34. The Standish Group International. Chaos Summary (2009). http://www1.standishgroup.com/newsroom/chaos_2009.php
35. United States Department of Labor—Bureau of Labor Statistics: Occupational Outlook Handbook, 2010–2011 Edition. Computer Software Engineers and Computer Programmers
36. Vitruvius: The Ten Books on Architecture (translated by M. H. Morgan). Dover Publications Inc. New York (1960)
37. Weigert, T., Weil, F.: Practical experience in using model-driven engineering to develop trustworthy systems. In: Proceedings of IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06), pp. 208–217. IEEE Computer Society (2006)

Author Biography



Bran Selic is President of Malina Software Corp. and also Director of Advanced Technology at Zeligsoft Limited in Canada as well as Visiting Scientist at Simula Research Labs in Norway. Prior to retiring in 2007, he was an IBM Distinguished Engineer responsible for defining the software tools strategy at IBM Rational. In 1992, Bran co-founded ObjecTime Limited, a company which successfully pioneered the use of model-based tools and methods in the real-time embedded domain. He has been involved in the definition and standardization of the UML modeling language.