International Journal of Engineering Applied Sciences and Technology, 2022
Vol. 7, Issue 6, ISSN No. 2455-2143, Pages 420-427
Published Online October 2022 in IJEAST (http://www.ijeast.com)

PERSONAL THOUGHTS ON AUTOMATION, PROGRAMMING CULTURE, AND MODEL-BASED SOFTWARE ENGINEERING

Mr. Vikas sawan
Assistant Professor Department of computer science and Engineering
Chandigarh University,Gharuan

Soumya upadhyay
Assistant Professor IT Department
College of engineering Roorkee(COER),Roorkee

Renu Bahuguna
Assistant Professor IT Department
College of engineering Roorkee(COER),Roorkee

Abstract: **When compared to more conventional development techniques, model-based software engineering (MBSE), a methodology for developing software, is characterized in part by much higher levels of automation. Many important components of development involve the use of computer-based tools, such as authoring support (many MBSE languages are primarily visual), automatic or semi-automatic verification, the automated conversion of requirements into corresponding programmers, and many more.**
There is little question that automation, when correctly designed and achieved, may significantly boost the productivity of software developers and improve the quality of their programme, given the historical examples, such as the advent of compilation technology. Therefore, it becomes sense to predict that MBSE will swiftly overtake other software development methods, much to how computer-aided design methods for hardware were quickly adopted. However, this hasn't happened.
This is an opinion piece that draws from the author's extensive knowledge of MBSE and its use in business. In it, we look at the factors that led to this paradoxical situation.

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

I. INTRODUCTION

Since the beginning of programming, computers have been used to facilitate the design and development of software. Given that computers are the ultimate automation machine and the main outputs of the development process are computers themselves that are stored, modified, and executed, it makes perfect sense.
Software development tasks including compilation, programme linking and loading, source programme generation and editing, version management, debugging, verification, documentation, and more have all been automated using computers. Compilation is among them, and it may be the most important in terms of its influence on output and quality. Third-generation languages, often known as high-level programming languages, were made possible by the development of compilers, which decreased the complexity of programme design by freeing programmers from having to worry about many technology-specific issues.
This allowed programmes to be specified using ideas and constructs that were much more closely related to human comprehension and the issue domain, which not only made it easy to port a given programme to a different system with little or no modification.
These significant advantages were soon acknowledged, and the great majority of practitioners swiftly transitioned from low-level to high-level language programming. Additionally, as programming grew more approachable, both the number of programmers and the variety of applications increased..
However, despite countless incremental improvements, the abstraction level provided by popular third-generation programming languages like C, Fortran, or Basic proved insufficient as the need for ever more complex and diversified computer systems developed.
Contributions to these languages. In particular, it was discovered that these languages' fundamental elements were frequently too fine-grained to allow the direct and

International Journal of Engineering Applied Sciences and Technology, 2022
Vol. 7, Issue 6, ISSN No. 2455-2143, Pages 420-427
Published Online October 2022 in IJEAST (http://www.ijeast.com)

understandable expression of many software applications' more complicated and domain-specific notions and relationships (i.e., their architecture).

This inevitably led to the employment of higher-order1 formalisms, such as finite state machines or entity-relationship structures, which naturally led to an increase in the level of abstraction of software specifications. (It's important to note that many of these formalisms were naturally pictorial in nature, as opposed to text descriptions, which are frequently more effective in explaining certain types of structures and relationships.) Now a common practise in the analysis and design of large software systems, the use of such higher-level formalisms.

But even though there is a clear parallel between this and the change in abstraction levels that occurred when languages were upgraded from the second to the third generation, and even though higher-level implementation languages like 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 widespread adoption of more contemporary implementation techniques and technologies. This has led to a growing semantic gap between third-generation programming language-specified software implementations and higher-level formalisms, which are often used to express software design specifications. Despite the fact that programming languages have advanced significantly over the past three decades, the fundamental level of abstraction (and, subsequently, the expressive

of the first third-generation languages is not noticeably bigger than that of the current main implementation languages, such as Java, C++, or C#. That is to say, it is nearly as challenging to identify the high-level architectural form and important design principles of a Java-written programme as it would be if the same programme had been written in Fortran or Cobol.

It goes without saying that this disparity in abstraction leads to issues that are frequently quite challenging to solve. There is a clear possibility that mistakes will be introduced during the informal conversion of high-level specifications into programmes, resulting in an implementation that does not faithfully reflect design intent. On the other side, it is also possible that a high-level design specification could recommend ineffective and even impractical systems by disregarding important implementation concerns.

Sophisticated software systems are best built through some kind of iterative and incremental approach, in which analysis, design, and implementation tasks advance either in parallel or cyclically follow each other in short succession. This reduces the possibility of such errors. Iterating between these notions becomes more challenging the bigger the abstraction gap between them is.

It would seem logical to use computer-based automation to help close the semantic gaps in this process. In reality, there have been many attempts to include such automation into software development, starting with so-called fourth-generation languages and continuing with the model-based software engineering used today (MBSE). Examine each in turn. of these in turn.

1 Fourth-generation computer languages

These high-level languages, often known as 4GL, are tailored for a particular application area or function. The Re-port Program Generator (RPG) language, a declarative language used to produce reports from databases, is an early example of a 4GL (International Business Machines 1964). When compared to analogous imperative-style systems written in a language like Cobol, it proved to be extremely effective for its intended purpose and significantly cut down on development time. Numerous more 4GLs have been developed and successfully used over time, most notably MATLAB (for mathematical and dynamic system modelling) (MathWorks 2008), SQL (for database queries) (Chamberlin and Boyce 1974), and SPSS (for statistical analysis) (SPSS Inc 2006).

The present emphasis on domain-specific (modelling) languages (DSLs) is the most recent illustration of this tendency. The main disadvantage of 4GLs is their extremely specialised nature, which is also what makes them profitable. These languages only have a small user base because of their narrow range of use. This typically implies that, in comparison to the far more prevalent general-purpose programming languages, it is significantly more expensive to provide very advanced automation support for such languages. The majority of the time, these languages' support tools are either in-house creations made specifically for the purpose or are offered by a small number of specialised suppliers (often just one). Utilizing internal tools necessitates allocating development and other resources to tool support. and development—resources that detract from core business. The cost of tools provided by a small number of vendors, on the other hand, tends to be higher and they frequently come with a significant risk of either the vendor or the products being discontinued. In addition, highly specialised tools and languages necessitate specialised education and abilities that are more challenging to acquire on the open market, suggesting higher training expenses.

On the other hand, there are numerous top-notch and interesting development tools that support popular general-purpose programming languages. Both the open source movement and tool providers with a focus on this type of product generate and support these tools. These tools are always getting better while becoming more affordable because to competitive pressures and the charitable goals of open source (many are available for free). Simply said, 4GLs cannot develop into an identical situation due to economies of scale. (Unfortunately, this is a factor that is frequently disregarded in current talks on the benefits of DSLs.)

International Journal of Engineering Applied Sciences and Technology, 2022
Vol. 7, Issue 6, ISSN No. 2455-2143, Pages 420-427
Published Online October 2022 in IJEAST (http://www.ijeast.com)

2 CASE tools

The unquestionable success of computer-aided design (CAD) tools, which significantly increased hardware design automation, served as the inspiration for CASE tools. CASE tools are an early attempt to directly convert higher-level formalisms used in software system analysis and design into similar code—an approach that seems without fault. They supported the creation of various analysis and design diagrams as well as usually some automated or semi-automatic code generation from them. Unfortunately, the success of CASE tools was never even near to that of their hardware counterparts. They are really frequently used as a paradigmatic example of a technology that promised a breakthrough but fell short.

It is important to look into the various factors that contributed to CASE's "failure"2 because they can provide guidance for any current or future attempts to use computer-based automation to close the gap between design and implementation.

The disparities in quality between design and implementation are one significant barrier. Design, especially in its early stages, is best done in an environment that is free from restrictions so that ideas can develop without being hindered by formality and pedantic precision. As a result, the majority of design languages supported by CASE tools tended to provide informal and imprecise formalisms. While implementation languages must be translated accurately and deterministically into deterministic programme code, they are required to be unambiguous and extremely formal. Any computer code produced using such formalisms was therefore lacking and needed to be completed by programmer-written code, which clarified any confusion and filled in any implementation-level gaps. Sadly, this severances the official connection between the code and the

that it was derived from. By accident or on purpose, the additional code may undermine or even go against the original design objectives. The automated return to high-level formalism might therefore no longer be possible,

The technique known as "round-trip engineering" has been used in various attempts to solve this issue (RTE). The code is reverse engineered into a corresponding high-level specification in this approach. But the most typical result of this approach is that the high-level specification gradually degrades into a graphical representation of the low-level code that is 1:1. 3 In these situations, a large portion of the value derived from a high-level representation is lost.

Additionally, compared to hand-crafted code, the code produced by CASE tools was frequently either triv- ial (therefore offering developers little real benefit), or of poor quality. Having little to no theoretical knowledge of or experience with the best ways to produce code from graphical formalisms contributed to this in part.

The proliferation of many high-level formalisms that appeared around the time that CASE tools were being developed was another issue with early CASE tools. For instance, there were around a hundred published analysis and design languages in the early 1990s (Graham 2001). Typically, a subset of these would be supported by each CASE tools, although seldom more than a few. Users were compelled into the vendor lock-in issues that were

In summary, CASE tools simply did not provide enough value to either the design-ers 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).

3 Model-based software engineering (MBSE)

In a way, MBSE is just a development of the CASE methodology. However, since the early days of CASE, there has been a large context shift, which significantly increases the viability of MBSE. Specifically:

The underlying technology have advanced significantly. This includes, in particular, more potent computing hardware (performance, memory capacity), as well as improvements in the design of modelling languages (the use of meta-modeling approaches), automated code generation techniques, and software tooling (the introduction of tool frameworks like Eclipse [Eclipse Foundation 2008]). in general

As our knowledge of the creation and application of modelling languages develops, so does our comprehension of both their accompanying issues and solutions. (However, it is accurate to remark that MBSE is still a long way from being a discipline of engineering that is well-established, that is, one that is founded on well-understood scientific and technical foundations.)

Unified Modeling Language (UML) by the Object Management Group (Object Management Group 2007a), which was developed and widely used, has significantly lessened the issue of the needless abundance of several high-level modelling languages and notations..

1. Numerous articles have been written about MBSE and its characteristics (e.g., Object Manage- ment Group 2003; Frankel 2003; Greenfield et al. 2004; Mellor et al. 2004). There is a lot of discussion around meta-modeling, domain-specific modelling languages, platform-independent models (PIMs), platform-specific models (PSMs), model translations, etc. According to this source, the central principle of MBSE may be distilled into two key ideas that we have already covered:

2. Numerous articles have been written about MBSE and its characteristics (e.g., Object Manage- ment Group 2003; Frankel 2003; Greenfield et al. 2004; Mellor et al. 2004). There is a lot of discussion around meta-modeling, domain-specific modelling languages, platform-independent models (PIMs), platform-specific models (PSMs), model translations, etc. According to this source, the central principle of MBSE may be distilled into two key ideas that

International Journal of Engineering Applied Sciences and Technology, 2022
Vol. 7, Issue 6, ISSN No. 2455-2143, Pages 420-427
Published Online October 2022 in IJEAST (http://www.ijeast.com)

we have already covered:
3. 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
4. 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 repre-sentation 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 rep- resents. 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 ab-straction 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, meth- ods, and expertise can be used throughout, thereby avoiding the qualitative discon-tinuities that characterize practically every other form of engineering development. Clearly, such a process, if properly supported by automation, has a much greater like- lihood of ensuring preservation of the original design intent. Furthermore, with suit- able 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) ex-pression of design ideas. As the process progresses, the degree of formality and con-sistency 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 im-portant feature that distinguishes good modeling languages from most programming

4. Pragmatic issues with computer automation in MBSE

It is evident from the above description that MBSE is not practical without ef-fective computer-based automation. And, although there are numerous examples of successful applications of MBSE in large industrial software projects, real-ized 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 re- quired 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 typi- cal MBSE design tool offers its capabilities via multiple overlapping categories of menu items grouped in unintuitive ways. In an often naive and simplistic interpre- tation of principles of GUI design, garnered mostly through ad hoc learning rather than systematic study, bizarre and confusing icons and graphics abound, suppos- edly 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 valu-ably 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.

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.

One promising approach to dealing with this problem might be to use an intel-ligent 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 ba-sic 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 in- adequate support for customization for specific application domains and environ- ments—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 con-figurations are defined for individual tools independently of other tools

International Journal of Engineering Applied Sciences and Technology, 2022
Vol. 7, Issue 6, ISSN No. 2455-2143, Pages 420-427
Published Online October 2022 in IJEAST (http://www.ijeast.com)

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 exis-tence 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 cer-tain properties (safety, liveness, performance characteristics, etc.). Unfortunately, in most cases this transfer is fraught with problems and requires some interven-tion. 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 with-out 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, inter- operability problems will remain.

Scalability problems. The abstraction power of models is needed most when deal-ing 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 in-dividuals 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.

This, of course, makes it extremely difficult to partition such a model into man-ageable 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.

5. 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 program-ming and to the type of personality that is attracted to it. One key element that distin-guishes programming from other forms of engineering design is its lack of physical impedance. That is, programming primarily involves the transfer of ideas into equiv-alent or near-equivalent specifications and does not require bending, lifting, or oth-erwise 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 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 se- ductive experience. For many individuals, the sense of personal gratification and mas- tery 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

International Journal of Engineering Applied Sciences and Technology, 2022
Vol. 7, Issue 6, ISSN No. 2455-2143, Pages 420-427
Published Online October 2022 in IJEAST (http://www.ijeast.com)

Marshall McLuhan (1964). One common undesir-able 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 consti-tute 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.

6. 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 check- ing 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 math- ematical 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 lan-guages. 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 ana-lyzing 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 execu-tion 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 pro-file of UML (Object Management Group 2008a, 2008b). Other types of quanti-tative 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 pre- cise 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 specifica-tion 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 con-fidence 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.

7. 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 capa-bilities 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, in- vention, 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.

II. REFERENCES

[1]. Brooks Jr., F.: The Mythical Man-Month. Addison-Wesley, Reading (1995). Anniversary edn.
[2]. 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.
[3]. Association for Computing Machinery (1974)
[4]. Eclipse Foundation: Eclipse documentation. http://www.eclipse.org/documentation/ (2008)
[5]. Ellsberger, J., et al.: SDL – Formal Object-Oriented Language for Communicating Systems. Prentice Hall, London (1997)
[6]. Frankel, D.: Model Driven Architecture – Applying MDA to Enterprise Computing. OMG Press, Indi-anapolis (2003)
[7]. Graham, I.: Object-Oriented Methods. Addison-Wesley, London (2001)
[8]. Greenfield, J., et al.: Software Factories. Wiley, Indianapolis (2004)
[9]. 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)
[10]. 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)
[11]. Magerko, B.: Adaptation in digital games. IEEE Computer 41(6), 87–89 (2008)
[12]. MathWorks: MATLAB Function Reference. http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html (2008)
[13]. Mellor, S., et al.: MDA Distilled—Principles of Model-Driven Architecture. Addison-Wesley, Boston (2004)
[14]. Mernik, M., Heering, J., Sloane, M.: When and how to develop domain-specific languages. ACM Com-puting Surveys 37(4), 316–344 (2005)
[15]. McLuhan, M.: Understanding Media: The Extensions of Man. McGraw-Hill, New York (1964)
[16]. Nunes, N.J., et al. (eds.): Industry track papers. In: UML Modeling Languages and Applications – «UML»
[17]. 2004 Satellite Activities, Lisbon, Portugal, October 2004 (Revised Selected Papers). Lecture Notes in Computer Science, vol. 3297, pp. 94–233. Springer (2005)
[18]. Object Management Group (OMG): MDA Guide, v.1.0.1. OMG document omg/2003-06-01 (2003)
[19]. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification,v.2.1.2. OMG document formal/07-11-02 (2007a)
[20]. Object Management Group (OMG): XML Metadata Interchange (XMI), v.2.1.1. OMG documentformal/07-12-01 (2007b)
[21]. Object Management Group (OMG): A UML Profile for MARTE: Modeling and Analysis of

Real-TimeEmbedded Systems, v.Beta 2. OMG document ptc/08-06-09 (2008a)

[22]. Object Management Group (OMG): OMG MARTE Information Day (June 2008b). http://omgmarte.org/Events.htm

[23]. Selic, B., et al.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)

[24]. SPSS Inc: SPSS 15.0 Command Syntax Reference, Chicago IL (2006)

[25]. Weigert, T., Weil, F.: Practical experience in using model-driven engineering to develop trustworthy com-puting systems. In: Proceedings of the IEEE International Conference on Sensor Networks, Ubiqui- tous,and Trustworthy Computing, vol. 1, pp. 208–217, 5–7 June, 2006