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

Bran Selic

Received: 21 July 2008 / Accepted: 1 August 2008 / Published online: 3 September 2008
© Springer Science+Business Media, LLC 2008

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

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

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

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

1 Introduction

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

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

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

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

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

---

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