Will MDD Fulfill Its Promises?

The theme of IEEE Software’s Sept./Oct. 2003 issue was model-driven development. MDD’s premise is that we can develop software through model transformation, from high-level models to programs. The authors presented MDD as a general-purpose concept or even a paradigm for future software development, with potential to dramatically improve productivity and change the way we develop software.

Models play an important descriptive role in software development. Plenty of successful model-based solutions exist in narrow application domains. Model transformations also have value, as CASE tools demonstrate. But there are reasons to doubt that model transformations will become the foundation of a general-purpose software development paradigm.

Almost 20 years ago, in his famous paper “No Silver Bullet,” Frederick Brooks remarked on the essential complexity of program structures that are meant for execution on a computer (programs and abstractions, such as UML models, alike). By raising the abstraction level, we can eliminate some of the accidental complexities but not the essential ones. If we abstract away essential complexities, our program abstractions become meaningless. They might have descriptive value, but they don’t define complete programs as is required for execution on a computer.

Raising the abstraction level has been successful in narrow domains, using domain-specific problem description languages (BNF for example) and generators (such as a compiler-compiler). Generators encode the implicit knowledge of the domain’s semantics. This implicit knowledge fills the gap between problem description in a domain-specific language and a program implementation. That’s why generators can offer efficient solutions in specific well-understood and stable domains but don’t scale up to a general-purpose programming paradigm such as MDD in which we can’t rely on domain-specific knowledge.

If MDD can’t rely on the domain-specific implicit knowledge, the essential complexity of a program we build must be explicitly expressed in models’ semantics. This is a big problem, especially when it comes to integrating partial specifications of problem semantics in different UML models (such as use cases, class diagrams, and so on). Will one semantic model exist for all UML models or will each UML model have its own? How much of models’ semantics can we hide from programmers, given that we can’t hide inherent complexity in general-purpose models?

One might hope that multiple models will automatically reduce complexity, by virtue of “divide and conquer.” After all, multiple descriptive models serve us well. But I don’t believe this will be the case. On the contrary, I believe that working with multiple formal models will be counterproductive. A tremendous amount of delocalized information and redundancies across models will exist. In descriptive, informal models, redundancy may be a virtue. But in formal, executable, and interdependent models, it’ll pro-
duce far more complexity than Brooks’ essential and accidental program complexity combined.

MDD’s essence is progressing from high-level to lower-level abstractions and code by model transformations. Given essential complexity, delocalized and redundant information, and unsolved model semantics problems, the technical feasibility of such transformations is unknown. Even if you find a solution, it’ll be exposed to a horde of new programs. During transformation-based development and maintenance, models will be in constant flux. We know how hard it is to change programs; the difficulty of indirectly fine-tuning programs through a combination of abstract models will be far greater. Our ability to change programs is also obstructed by obscure mappings between the reasons for the program change and the affected program components. One source of change (say, a change in user requirements) might affect multiple components, particularly if it affects component interfaces or global system properties. The impact of changes is mostly undocumented, and I don’t see any UML mechanism to enhance their visibility. Multiple formal models will make change far more difficult than in the case of programs. A change might have to be propagated across multiple models, not forgetting redundancies that’ll have to be noted and properly updated. It’s unclear what kind of mechanism might enable formal model transformations and change propagation.

Finally, propagating changes backward from lower- to higher-level models is an issue. What happens if I modify a lower-level model? Either I can propagate changes to the affected higher-level models or my models will become disconnected. Propagating changes backward from specific to more abstract representations is notoriously hard in much simpler situations. In the context of formal models, difficulties can only multiply.

There is a wealth of experiences from automatic programming research of the 1970–80s and CASE tools of the late 1980s. Those experiences point to critical problem areas that are also relevant to MDD and can be used to validate its basic assumptions. The software industry has gone through many cycles of technological hyperbole. Each one always starts with overstated promises and a lack of hard evidence. Hyperbole is more socially and psychologically satisfying than technical phenomena. Often, premature ideas get surprisingly strong support from major industry players, possibly to convince customers about their commitment to the newest computing developments. For researchers, hyperbole often opens new sources of grant money. So what’s wrong with hyperbole? Only that we’ve “been there, done that” to our detriment too many times. It’s time for software engineers to take a mature, evidence-based approach to supposedly new ideas.

Stan Jarzabek
National University of Singapore
stan@comp.nus.edu.sg
Stephen J. Mellor responds:

Mr. Jarzabek raises several interesting technical points that, summarized ruthlessly, boil down to just two: models are insufficiently precise ("overly abstract") to transform reliably into systems without a developer’s interference, and models’ multifaceted nature, in contrast to the block-structured nature of today’s programming languages, introduces consistency and change-control issues. He’s right.

He’s also correct to identify “integrating partial specifications of problem semantics in different UML models (such as use cases, class diagrams, and so on)” as being the “big problem.” This problem exists because UML is defined as a set of separate “models”—diagrams, really—that don’t necessarily fit together to form a coherent, executable whole. In turn, this leads to “a tremendous amount of localized information and redundancies across models.” I agree wholeheartedly.

But this sorry situation isn’t inherent in MDD. You can address both problems by defining a single underlying ground truth against which all diagrams are merely views. A change in one diagram, such as adding a signal send to a state chart diagram, would then necessarily be reflected on the diagram that shows communication between state charts. Ground truth, a concept from game simulation that captures the notion of what’s really going on irrespective of players’ imperfect perceptions, is necessarily internally self-consistent, and it’s one source for translation into systems. (The other source is a set of translation rules that are made explicit in MDD and can be controlled by the developer.)

The executable-modeling community recognizes these flaws, and efforts are underway to define a standard, simplified, interchangeable ground truth that captures the executable essence of a single subject matter that can be woven together with others. This executable metamodel forms the basis of a tool chain of model compilers, checkers, and analyzers of various kinds. Each tool in the chain deals with ground truth, a tractable subset of UML semantics. Developers interact with subsets of the concrete syntax of UML for different users and methods and—more importantly—with domain-specific languages tailored to a user’s domain.

In Mr. Jarzabek’s last paragraph, he says there’s a “wealth of experiences from automatic-programming research of the 1970–80s and CASE tools of the late 1980s” but doesn’t enumerate any of them. At the same time, he says “[i]t’s time for software engineers to take a mature, evidence-based approach to supposedly new ideas,” another idea to which I can only roar assent.

There are tools today that define an executable profile and use it, in industrial settings, to accrue the benefits he claims are hyperbole. Why, then, do other authors in this IEEE Software issue call these working systems a “cocktail-party myth”? I can’t tell you; I can only point to the benefits, the decreasing costs derived from a standard tool chain, and working systems. Today. And that’s no hype.