Model-Based Development: A New Approach to Engineering Medical Software

Arnab Ray and Raoul Jetley

With an increasing number of medical device features being implemented in code, the amount of software that is present in a modern device as well as its complexity and criticality has grown sharply over the years. Existing quality-control regimes for software, dependent as they are on traditional inspection and ad-hoc testing techniques, has been unable to meet many of these challenges. The numbers tell the story. In 1998, close to 8% of device failures could be traced to software errors. Currently, the number of device recalls due to software problems is believed by some to be about 18%.

Model-based development (MBD) is often suggested as a candidate solution, a novel way of doing software development and quality control.

So what is model-based development and what it does it mean for professioals working in the medical instrumentation field?

Read on.

The Evolution of the Programming Language

A long time ago in a land not so far away, almost all code was written in assembly language. Being essentially a sequence of loads, arithmetic operations and stores, assembly code was difficult to understand, produce, and maintain with almost all but the most basic design structure of the program being obfuscated by the low-level nature of the microprocessor's native notation.

"MBD has attained increasing popularity in the aerospace and automotive industry because of how it supports the production of better software.

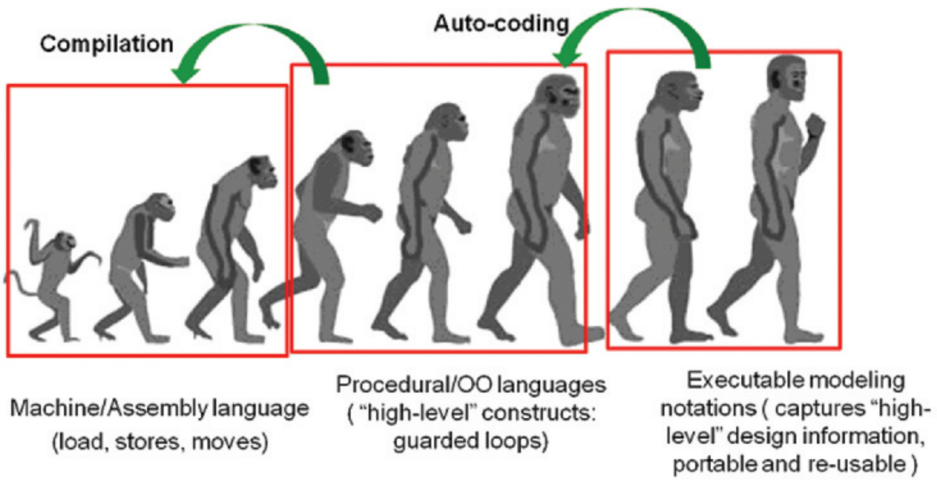
A new era was ushered in with the advent of the programming language compiler. The compiler allowed engineers to abstract away low-level assembly instructions, enabling them to program using higher-level logical constructs like guarded loops. Since the formalism of a programming language was more akin than assembly instructions to the way an engineer conceptualized a system, the code could be better understood, developed, maintained, and debugged. Microprocessors, of course, still understood only machine language and this is why compilers were were needed to transform higher-level procedural code into lower-level "machine understandable" instructions. Compilers, thus, isolated the engineer from details of the underlying microprocessor and enabled easier porting of an application across microprocessor families.

In the next stage of the evolution of the programming language, object-oriented (OO) languages like C++ and Java attained primacy. A principal feature of these languages was that they allowed the system to be built by defining abstract behavior and relationships between conceptual entities (called "objects"). This made the engineering of software easier as it further bridged the conceptual gap between design and implementation. However, design and implementation were still not totally decoupled. For instance while using C++, developers not only had to think of design aspects like behavior of objects but also manipulate implementation artifacts like pointers and memory.

MBD[1, 2, 3] bridges this final gap between design and implementation. MBD is a formal methods-based software development paradigm that uses the notion of executable models to design and verify software. Executable models may be looked upon as the latest stage in the evolutionary

Arnab Ray is a research scientist at the Fraunhofer Center for Experimental Software Engineering and also an assistant adjunct professor at the department of Computer Science, University of Maryland College Park. Email: Aray@fc-md.umd.edu

Raoul Jetley is a research scientist at the U.S. Food and Drug Administration's (FDA) Center for Devices and Radiological Health/Office of Science and Engineering Laboratories. Email: Raoul.Jetley@fda.hhs.gov


Compilation
Auto-coding
Machine/Assembly language (load, stores, moves)
Procedural/OO languages ("high-level" constructs: guarded loops)
Executable modeling notations (captures "high-level" design information, portable and re-usable)

path of programming languages. Such notations allow for the explicit capture of data-flow (through system models) and control-flow (through states and transitions) thus enabling engineers to program using exclusively conceptual design-time artifacts. Moreover, unlike traditional design models, these can be simulated just like code (hence the word executable).

Design executability affords engineers with an opportunity they never had before–the ability to debug designs in the same way that they can debug code. In fact, models allow them to go a few steps further. Mathematical theories of executable modeling have been formulated based on the run-time semantics of models. This enables the construction of automatic proofs of design correctness allowing engineers to make assertions on the code. These assertions usually enforce that the software is assured to always work correctly, regardless of what the inputs are. This is a much more rigid guarantee of proper operation than that afforded by conventional testing approaches.

In the same way that a compiler converts procedural/object-oriented languages into a form that can be run by a microprocessor, code-generators can produce code in C/C++ from executable models. This not only leads to a clean distinction between design aspects (architecture, control flow, data flow) and implementation (memory and resource management), but also allows engineers to port their designs over multiple target platforms by plugging in different code generators.

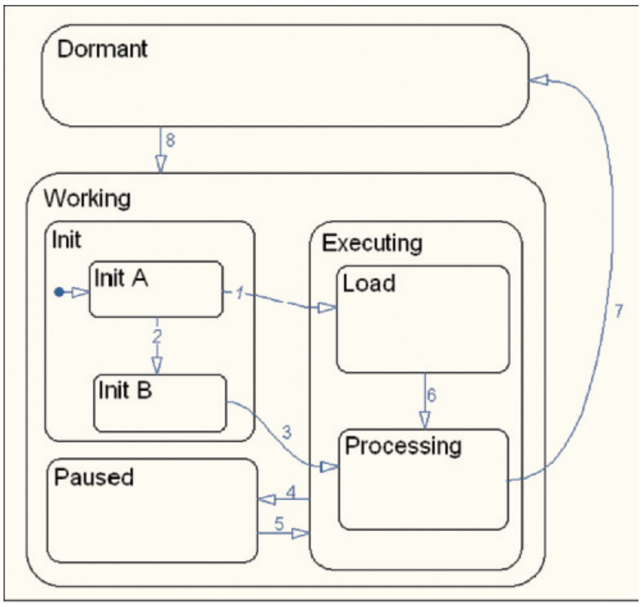
Benefits of Model-Based Development

MBD has attained increasing popularity in the aerospace and automotive industry because of how it supports the production of better software. The medical device industry has many concerns similar to these industries in that they too rely critically on the proper operation of software, growing increasingly complex by the year, to ensure patient safety. In this context, we envisage a similar uptake in the adoption of MBD driven by needs in three areas:

1. Rapid prototype construction: Device manufacturers spend a considerable amount of time and effort on human factors engineering, studying the interface that a device provides to patients and care-givers. Executable models, being simpler than code and having many of the low-level aspects of code such as memory management abstracted away, can be constructed much more rapidly than concrete implementations. Because of this, executable models can be used to construct rapid device prototypes which may be simulated in software and deployed to study different design choices for the user-interface and the ramifications of those design choices on the final product. Additionally, executable models are typically rendered in graphical hierarchical notations and these visual artifacts, expressed in a more abstract manner than code, allow for better system comprehension.

2. Software architecture specification: The software architecture, like a building's floor plan, lays out the structure of the system—the software components, the communication between them, and the constraints on component interaction. (For instance, in a layered architecture a restriction might be that a component cannot invoke any other component not immediately in the layer below it.) In non-MBD based approaches the software architecture is typically specified using textual or informal notations. Because of this, it becomes very difficult to automatically ensure that the implementation respects the architectural constraints laid down as part of the design rules. Consequently "architectural degeneration" of a software system often takes place over multiple development cycles ultimately leading to the code deviating so drastically from original designer intent so as to be virtually unmaintainable.

MBD forces engineers to express their designs in a well-defined notation. As a result, architectural information-like component and connector definitions is captured in a much more formal manner than would be



A simple executable model expressed as a hierarchical state machine

possible in non-MBD workflows. Automatic code-generation allows the implementation to maintain the design architecture, thus ensuring conformity of implementation to design structure.

Sometimes, however, the situation is made more complicated by the need to add code manually after code-generation. This could potentially break the architecture. To make sure that this does not happen, the final software architecture is extracted from the code using reverse-engineering techniques and compared to the design architecture. If deviations are detected, then changes need to be made at the design or the implementation level so that both the representations may remain in consonance with each other.

3. Design verification: As mentioned previously, the formal execution semantics of modeling notations allow for very precise definitions of how the software behaves. This makes it possible for researchers to develop theories, algorithms, and tools for automatically verifying designs. In the technique known as model-checking,[4] requirements are formulated as mathematical formulae and the design model is then explored, using sophisticated graph traversal techniques, to check whether the design satisfies the requirements. In instrumentation-based verification,[5] requirements are formulated as mini-models themselves and harnessed to the main design models. Tests are then exhaustively generated by an "intelligent" (automated) pessimistic tester which tries to execute the model in such a way that the requirements are violated. If the tester fails (i.e. does not succeed in "breaking" the model), then the requirements are satisfied.

Model-based development can benefit medical software engineering in multiple ways. By allowing engineers to develop executable designs for their systems, it provides a framework for formal verification and validation. Executable models, while more abstract than programming languages, can be simulated, and used to automatically derive source code, thus promoting re-use. The models themselves can be behavioral (or function based), object-oriented, or defined as a state machine. Though much of the mathematical analysis has traditionally been used only in academic communities, of late a number of commercial tools that leverage the power of executable modeling have become available. This has made it feasible to use model-based development in a realistic medical device engineering environment.

Currently, researchers at the Fraunhofer Center for Experimental Engineering and the U.S. Food and Drug Administration (FDA) are working to pilot model-based development techniques for medical devices. Models of a generic infusion pump have been constructed with respect to a set of software requirements and the models verified against these requirements through techniques like instrumentation-based verification.

It must be remembered, however, that MBD, like any other development technique is not a panacea. While it may help software developers verify the system against requirements, it cannot assure that the requirements themselves are correct. More importantly, using MBD does not imply that the source code need not be validated. The technique is best used as a complement to, not a replacement for, traditional verification and validation technologies. ■

References

1. **R Kampfner**. Model-Based Development of Computer-Based Information Systems, Proceedings of the Workshop on Engineering of Computer-Based Systems, (ECBS) pp.354, 1997
2. **D. C. Schmidt**. Model-driven engineering. IEEE Computer, 39(2) 25-31.
3. **R France, B Rumpe**. Model-driven Development of Complex Software: A Research Roadmap, Proceedings of the International Conference on Software Engineering (ICSE), pp. 37-54, 2007.
4. **S Merz**. Model checking: A tutorial overview, Modeling and Verification of Parallel Processes, Lecture Notes in Computer Science , vol. 2067, pp. 3-38, 2001.
5. **C Ackermann, A Ray, R Cleaveland, J Heit, C Shelton, and C Martin**. Model-based design verification: A monitor based approach, Society of Automotive Engineers (SAE) World Congress, 2008.