Bruce Powel Douglass is a writer with an attitude. He is passionate and enthusiastic about real-time systems, safety and reliability, and direct execution of models. In Texas, to where I’ve just moved, attitude, passion, and enthusiasm are a part of the culture. Your mileage, as they say, may vary.

“Safe enough’ looks different at 35,000 feet.”

The Boeing 777 has been described as a million lines of Ada flying in tight formation. It is a completely fly-by-wire system. No mechanical or hydraulic linkages sit between the pilot’s controls and the control surfaces, just those lines of code. I’ve got an attitude, too, about the safety and reliability of that code.

In spite of the subtitle, this book is relevant to most software developers, not just real-time developers. Even though embedded systems are the most common kind of computer (there are more microwaves and VCRs than mainframes and PCs), true real-time systems are not common. Mostly people want real fast. And they want it real soon. Most of the problems addressed by this book are present in modern applications with GUIs and Web or network connectivity. Only the sections on schedulability apply exclusively to real-time systems.

Although not included in the subtitle, process is a big part of this book. Douglass pushes process but is also realistic about it (for example, architectural design can be skipped in small systems). He lists the relevant IEEE standards and says that most can be fulfilled in a couple of pages. The purpose of process is to ask the right questions at the appropriate time, not fill a cabinet with “objective evidence” for auditors.

Douglass’s flavor of process is Rapid Object-oriented Process for Embedded Systems (ROPES). If you’ve been following OOA/D development, there is nothing terribly radical here. He puts a heavier emphasis on safety than most treatments. He is an advocate of code generation from UML, so his usage of UML is more complete than most. UML statecharts, a kind of state diagrams derived from Harel statecharts, get a lot of coverage. This is because they fully specify the behavior, not just the interfaces. UML statecharts can be executed and debugged at the UML level. Class diagrams, on the other hand, can be used to generate header files but not executable code.

Some of his ideas have me jumping up and down, saying, “yes, that is what I’ve been trying to do.” Others do not. For
example, I am not yet convinced of the value of the direct execution of models (for example, simulating the system inputs and watching an animation of the state diagrams). A lot depends on the quality of the tool.

UML is a language and as with any language, it takes some time to develop fluency. The book includes a whole section of statechart patterns that are worthwhile in their own right and as practice to develop fluency. Most patterns I’ve seen have been class diagram patterns. This is the first time I have seen patterns of temporal behavior. It’s one of the ideas that made me say, “yeah, that’s it.”

I recommend this book with a caveat, however: it’s full of cynical footnotes. For example, the footnote to the sentence “One problem that arises with environmental interaction is that the universe has an annoying habit of disregarding our opinions of how and when it ought to behave” reads “It is like teenagers in this regard.” In a previous job, I sat next to Larry McAlister, one of the peer reviewers for the book, while he was going over the proofs. I sometimes wish he or one of the editors had deleted even more of the cynical footnotes. Too much cynicism leads to apathy, not action.

One person’s cynicism is another person’s truth. One of the book’s diagrams includes Translation Defects (for example, compiler bugs) as an input to the development process. Is this true? Unfortunately, yes. Is making this explicit useful? Only as far as naming one of the many fudge factors that go into making an estimate.

The size of this book—750 pages—is both a strength and a weakness. Reading it takes a week or two. If you read it at bedtime, it will take even longer. This is dense, deep stuff. On the plus side, the extended length allows multiple examples and views of the important points. My understanding of several key points of UML and OOA/D deepened while reading this book. Multiple views (for example, architecture, analysis, and design class diagrams) of the underlying model are included, not a single class diagram that is elaborated over time.
The book comes with a CD-ROM that includes demos for Rhapsody, a “fully constructive UML Visual Programming Tool” and TimeWiz, an “integrated tool for timing analysis.” The former is a competitor to Rational Rose. The latter performs timing and schedulability analysis. Douglass is the Chief Evangelist for i-Logix, the developer of Rhapsody. Both demos are crippled versions that will not save your work. I walked through the tutorial for Rhapsody. It looks promising. The statechart animation and debugger is by far the most interesting feature. I don’t have one of the supported C++ compilers (Visual C++ 4.0 and 5.0, Borland C++ 5.0.2, Tornado 1.0, pRISM+ PPC 2.2.2, or pRISM+ 2.2.6) and was unable to try it out. The installation notes indicate that the demo version does not have statechart animation, only the high-end development package. Disappointing choice.

Both Rhapsody and TimeWiz require a significant investment of time to use properly. I suppose you could enter an already worked out project just to evaluate them. I’d want to road test something like this over the lifetime of a small project before committing a full-blown project to it. My suggestion to manufacturers is limit the size of the model, not the capabilities or use time. It isn’t many projects, even toy or personal ones, that can be completely developed and the documentation updated in 30 days (or however long you can keep the program running without exiting).

Overall I like this book and intend to consult it on a regular basis. It is a book for working software developers (people who are already “doing hard time”), not undergraduate computer science students. Following its advice is a lot of work up front to save time in the long run. To appreciate it, you have to have already experienced that “quick and dirty” isn’t quick.

The mark of a great book is how it holds up over the long haul. By this measure, Doing Hard Time is a very good book. Having read it through once, it now sits on the bookshelf next to my workstation for road testing. It’s going to take a year or two of use to fully absorb. My preliminary judgment is that if you write software with multiple concurrent inputs and you are looking for a better way to build software, read this book.

Jeffrey L. Taylor has been a software engineer for over 25 years. About half of this time has been devoted to embedded and real-time systems. His current research areas are open source software, Linux, and software development. He may be reached at jeff.taylor@ieee.org.

Jeffrey L. Taylor has been a software engineer for over 25 years. About half of this time has been devoted to embedded and real-time systems. His current research areas are open source software, Linux, and software development. He may be reached at jeff.taylor@ieee.org.