Failure Modes and Effects Analysis during Design of Computer Software

Nathaniel Ozarin, The Omnicon Group, New York

Key Words: FMEA, Software FMEA, Software failure, Mission critical software, Software fault tree

SUMMARY AND CONCLUSIONS

Performing FMEA on computer software presents problems and challenges not found in FMEA of electronic hardware. Contractual directions are usually very limited or nonexistent, leaving the analyst to establish and tailor guidelines needed for a particular analysis. Where code is unavailable or off limits to the analysis, the FMEA is of limited usefulness but can still contribute to a more reliable system design. Unfortunately, many reliability analysts will have more difficulty developing an approach to software analysis than doing it. An understanding of the software design process and a discussion of various approaches to software design FMEA is presented to make development of an appropriate approach and performance of the analysis itself easier to understand.

Moving from the lowest level of analysis to the highest level – typically from the method level to the module or package level – a FMEA becomes less accurate, less precise, and less informative, while the process becomes less difficult, less tedious, and less time-consuming. Moving from the lowest level of analysis to the highest also means a FMEA is based increasingly on the stated intent of the software designers and less on the actual product behavior. For any analysis above the code level, the analyst's conclusions about effects at each level will unfortunately be no better than the descriptions that the software designers provide.

1. *INTRODUCTION*

Aerospace system developments sometimes require a FMEA to be performed in the early stages of computer software design to help avoid unpleasant surprises with the finished product. Unfortunately, the contractual requirements for FMEA on software are usually very vague, leaving the software development management to decide when to perform the FMEA, and leaving the analysts to figure out how to perform it. Both groups need to determine an approach whose goals are to remain within budget, keep the customer satisfied, and provide an analysis that is genuinely useful.

While is it naturally desirable to perform failure analyses as early as possible in the design process, the accuracy and completeness of the analysis can be no better than the material on which it is based. When an analysis focuses on software design rather than implementation in code, the available material is usually sketchy and ambiguous because the designers know that anything they write during the design stages will be heavily revised or completely rewritten after the design has been implemented. The design is also highly subject to change because software development is a very fluid process during which designers can (and continually do) make beneficial changes without penalty to their development schedule. It is therefore important for analysts to understand the classical software development process and its many modern variations, and how the processes are commonly stretched and twisted to meet schedules.

There is no generally accepted process for performing a FMEA on software designs that exist only as diagrams and descriptions. Moving from the highest level of analysis to the lowest level – typically from the module level to the code level – a FMEA becomes more accurate, more precise, and more informative, while the process becomes increasingly difficult, tedious, and time-consuming. Moving from the highest level of analysis to the lowest also means a FMEA is based less on the stated intent of the software designers and more on the actual product behavior.

In any particular analysis, the smallest part that can fail is defined by the analysis level – for example, in an analysis at the software object level, the lowest unit that can fail (i.e., that can produce unexpected inputs to other units) is the object – but the resulting failure effects ripple upward and may also appear at the module level, and they will certainly appear at the system level.

2. *WHAT SOFTWARE FMEA IS AND ISN'T*

Software FMEA is a means to determine whether any single failure in computer software can cause catastrophic system effects, and additionally identifies other possible consequences of unexpected software behavior. Software FMEA does not predict software reliability, but aims to determine whether the failure of any single software element can cause specific catastrophic events or other serious effects. At the same time, the analysis can identify possibilities of less serious consequence so that the design code can be made more robust in specific areas before deployment. In a software FMEA, a failure is a software variable that is assigned an unintended value. This kind of failure can occur in many ways – for example, when a memory location is unintentionally overwritten, when internal processor or memory circuits fail, or when bad data is received from the outside world. The analysis seeks to determine observable

system effects – usually manifest via system hardware and therefore dependent upon hardware analysis – when any one software failure occurs, and in particular to determine whether any single software fault can result in a catastrophic event.

The software FMEA looks for consequences of *all* potential software failures without trying to determine the causes of the failures. It is independent of two essential but different kinds of analysis: (1) how the software design meets requirements, and (2) the adequacy of the requirements themselves. A subsequent analysis considers the design or the implementation. At the code level, this means analyzing effects of each variable when it takes on an unexpected value. For analysis at a higher level, such as the software object level, this means analyzing variables passed among the objects. The FMEA also does not consider correctness of algorithms or problems resulting from real-time design errors, but makes the assumption that every variable passed between software units might fail without regard to cause.

Table 1. Summary of the Software FMEA Process

Step	Subject	Description
1	System and Software Familiarization	Using tools and guidelines to understand the system under analysis.
2	Database Tool Development	Development of linked tables to maintain information and guide analysts.
3	Developing Rules and Assumptions	Applying knowledge and experience to lay out clear rules for analysis.
4	Developing Descriptive Failure Modes	Defining the ways that software units can be fail, and establishing failure causes.
5	Determining System Effects of Individual Failures	Examining software units one by one while using data in previously developed tables to aid the analysis.
6	Generating the Report	Using the database tool to automate report generation.

The complete software FMEA process using a database tool is summarized in Table 1 (adapted from Ref. 3). The steps in the table all apply to software FMEA, but the table does not address the software development process or the relative merits of analysis at different stages of development. The table also does not address how one determines what parts of a software development should be subject to FMEA.

3. *THE SOFTWARE DEVELOPMENT PROCESS AND FMEA*

Books and articles on software development processes can fill a library and developers are continually developing new ways to build software better-faster-cheaper. Fortunately, it is possible to summarize the process basics to the extent that it affects FMEA. Table 2 lists the logical, classical steps to software development in which each step must be completed – and checked against lists of specific accomplishments and required products – before proceeding to the next step. In the real world, however, a development can be completed only by bending and stretching the process. Many commercial software developments deliberately break the process and still manage to succeed (never mind that the resulting product may well be failure-prone and unmaintainable) but aerospace developments generally require developers to adhere to a detailed process-based plan. How can developers bend these steps in a highly-structured aerospace environment where detailed requirements are dictated by a customer-approved development plan, and what needs of the FMEA analyst do the steps fail to address?

First, it makes no sense to hold up an entire development because some requirements need further refinement. Software developers therefore apply the Table 2 steps independently to different parts of the software, where each part has its own set of steps and its own schedule. One section of the design may be stuck in the requirements stage while another section with better-understood requirements may be in the detailed design stage. Developer judgment further blurs the lines between steps when the designers decide that incomplete requirements affect only a relatively isolated part of a software item and it is safe to proceed with the rest of that item's design.

Second, the steps in Table 2 are highly iterative, sometimes to the extent that they are performed simultaneously. The classical descriptions of the Table 2 steps recognize that iteration is both inevitable and natural, but iteration – going back and taking time to revise requirements and designs – has a time and dollar price and developers tend to put off revising documents as long as possible. Sometimes documents are never properly revised. There are two main consequences here: one, documentation is always out of date during design stages, and two, when it is finally revised, it's often revised from memory and will be less accurate than when the changes were first made in someone's head.

Third, the documentation prepared during the design process is often heavy with diagrams and pseudo-code, and light on text. Pseudo-code, or program design language (PDL) are vague terms applied to descriptions of detailed

Table 2. Classical Software Development Process

Phase	Title	Description	How Documented
1	Requirements Analysis	Understanding and organizing customer requirements, deriving new requirements as needed	Database tools or paper documents
2	Requirements Allocation	Assigning each requirement to one or more parts of the software design	Database tools
3	Architectural Design	Developing high-level design units, communication needs	Design tools
4	Detailed Design	Pursuing design details to the coding phase	Design tools
5	Implementation	Writing, debugging, and integrating code	Software source code
6	Test	Testing high-level and system-level functions and performance	Usually paper documents, initially

software operations presented as step-by-step statements that should be in plain English sentences and phrases. Unfortunately, pseudo-code in documentation at any design stage often looks like real computer code, and sometimes it *is* actual code put there in an attempt to meet documentation requirements. Finally, document authors are generally development team members who naturally consider their own needs (and document requirements) rather than address needs of outsiders concerned with reliability.

Fourth, design-stage documentation generally focuses on what the software does during normal operation rather than how it handles unexpected problems. This emphasis occurs in part because handling of unexpected situations such as dividing by zero, illegal attempts to access memory, or dealing with out-of-range input data is usually handled at the code level as a matter of good code design and good code standards. Software exception and error handling is rarely part of design-stage documentation.

These four considerations mean the analyst facing software FMEA of an in-process software design should expect several things.

(1) At any stage of software development some parts of the software will be less mature than others, where 'mature' means, for FMEA purposes, that descriptive source material (including verbal discussions with the developers) is probably less likely to be ambiguous and incorrect, and less likely to change as the design proceeds. Since the analyst must work with one set of baseline documents, the software manager must understand that the FMEA, upon completion, will have been based on obsolete information.

(2) Documentation will generally be vague, ambiguous, and often incorrect. As a result, the analyst and the software manager must expect the developers to consider and answer many questions. Seeking developer assistance can be politically delicate (or worse) because the developers have their own problems and deadlines, and because some answers may not exist yet. For the same reasons, getting written answers to your questions may not be possible. The analyst must therefore be prepared to make some assumptions about software behavior (and state the assumptions in the analysis) where answers cannot be obtained.

(3) If the task is to perform a FMEA at a level higher than the code level, then the analyst should not be looking at pseudo-code if it is in fact real code, or something very close to it. Examination of code should be beyond the scope of a high-level analysis, unless the terms of the effort establish otherwise.

4. *GRAPHICAL REPRESENTATIONS OF SOFTWARE DESIGNS*

Large, modern software projects are usually developed using software tools to help designers graphically represent requirements and the evolving design. The diagrams produced with these tools – if designers put the effort into them – are an excellent basis for software FMEA. Unified Modeling Language (UML) is currently the most widely used modeling mechanism for capturing system design structure. The following four sections describe four of the most widely used kinds of UML diagrams. These descriptions are intended as introductions and are major oversimplifications of real UML diagrams. Ref. 1 gives an excellent detailed description of real UML notation and usage. Note that UML is a visual design representation that does not address code implementation – that is, it won't tell you anything about the programming language selected to implement the design or details of coding.

4.1. *Use case diagrams* are among the first diagrams developed by the software development team. These diagrams illustrate software functionality at the highest level by echoing functional requirements. Functional requirements establish what the software does, as distinct from performance requirements, which specify (for example) how accurately or quickly the software does them.

A use case is drawn as a bubble or oval whose label should be a meaningful description of that use case's functional requirement – for example, Provide User Display (Figure 1). A use case may be shown with connecting lines or arrows to other use cases that expand upon the details of its functionality – for example, Provide User Display may be surrounded with connections to other use cases labeled Provide Sign-on Display, Format Error Text, and Resize Graphic Panels. Finally, use cases show interactions with outside things such as external software, hardware, databases or people. Outside things are represented at a high level as stick figures called "actors," although they usually don't represent people, and connecting lines or arrows to actors show interactions. In sum: use case diagrams show, at the highest level, *what* software does and the external things with which the software interacts, but not *how* it does things.

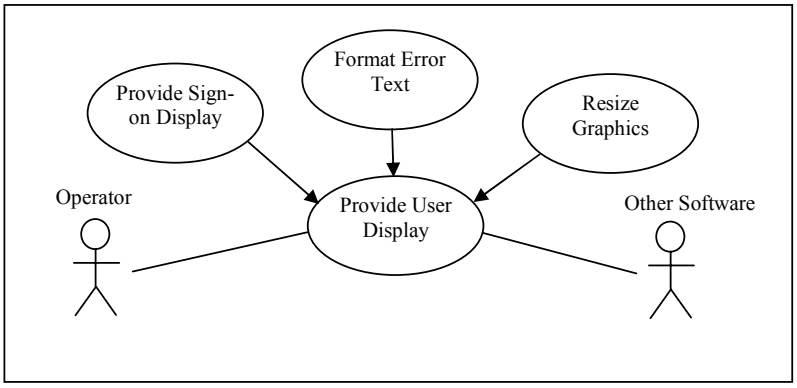

Figure 1. A simple use case diagram, "Provide User Display."

Use case diagrams become particularly useful for design development because they are generally accompanied by descriptive introductory text and additional text describing each use case. The text should include a step-by-step sequence of very brief events that explain how each use case operates. For example, imaginary use case Resize Graphics might include these steps: (1) Get graphics item from user

preferences, (2) Determine graphics window size based on graphic complexity other data with higher display priority, (3) rescale display items, (4) Send results to graphics buffer. Such steps focus on *how* use case requirements are approached and are the beginning of the software design.

4.2. *Sequence diagrams* pick up where use case diagrams leave off by adding new information to a use case's sequences of events. In general, each use case will have an associated sequence diagram, although a use case representing a simple item of functionality may not need a use case to explain how the associated software works, and a use case with more complex functionality may require several sequence diagrams to illustrate the events needed to make the function happen. Figure 2 is an imaginary sequence diagram. Software units that usually become software objects are shown as rectangles with tails. Event sequences are shown as interactions among objects (and externals) by descriptively labeled arrows. The object at the tail of the arrow initiates the event, and the object at the head of the arrow reacts to it. The labels in time become method names (function calls) between objects. As with use case diagrams, designers generally supply descriptive text to briefly explain each event. Sequence diagram development furthers explains the software design.

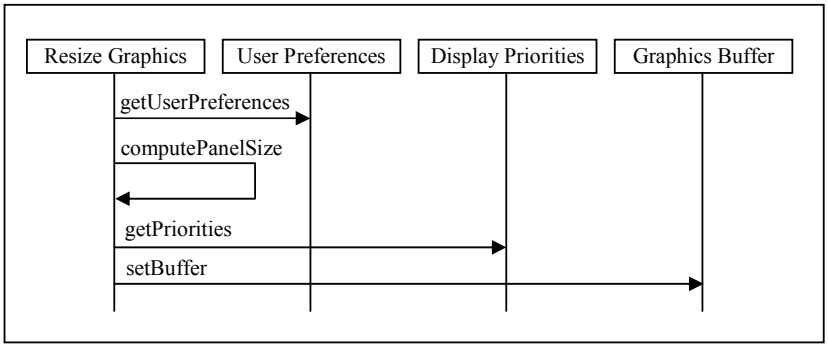

Figure 2. A simple sequence diagram, "Resize Graphics."

4.3. *Class diagrams* bring together information from one or more sequence diagrams by putting all methods belonging to an object– which may appear over several different sequence diagrams – into one location on a diagram (Figure 3). Although "class" and "object" are often used interchangeably in conversation, there is an important distinction: a class is a definition of a particular software unit with methods (function calls) and attributes (variables), but when the methods and attributes are assigned to and occupy memory where they can be do something useful, the working unit is called an object. For example, one engine class may be made into four engine objects in memory, with each behaving identically.

A class diagram generally shows several related classes and indicates interclass communication. A class is shown as a rectangle in which all class methods (function calls) are listed in one part of the rectangle, and attributes (variables) are shown in another section. A labeled line between two classes is called an association and indicates that there is communication between the classes, although the direction of the communication and its contents isn't always obvious from the label. The label's meaning is also usually at too high a level to be useful for FMEA purposes. Associations may also show interaction with external elements such as groups of software classes called packages. Packages are represented by other labeled symbols, usually rectangular and perhaps shaded to distinguish them from classes.

Text accompanying classes should summarize what the class does but should also contain detailed steps explaining how each class method works. These steps are often in some form of program design language (PDL). In a perfect world, class diagrams and their associated text should be sufficient for programmers to take over. Unfortunately, during the design process, and even after, the information will be neither complete nor correct, but should give a pretty good idea of how the implementation works.

Real class diagrams also include attribute data types, names of arguments passed to methods and their data types, return data types, plus other important information not shown in Figure 3 due to space limitations.

4.4. *Architecture diagrams* show the big picture – major system components as hardware or software, external items, component communications, and any other free-form information that designers wish to include. Architecture diagrams are often developed early in the design to clarify early system concepts and guide development of other UML diagrams.

In addition to the four diagram types introduced here, UML includes several other diagrams, notably constraint diagrams that show performance requirements along with their associated design parts, and data flow diagrams. From the FMEA viewpoint, the more diagrams, the better.

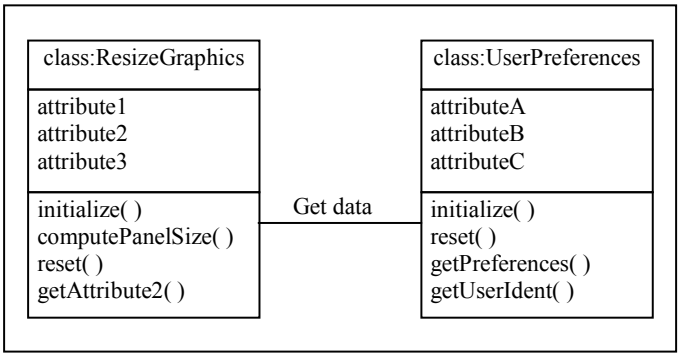

Figure 3. A simple class diagram.

5. *BRINGING IT ALL TOGETHER*

The phases of classical software design phases and the major UML diagrams seem like two different worlds, but there is correspondence between them. Table 3 summarizes groupings of software units from highest (package) to lowest (method). These groupings are not emphasized in the classical design phases – although they evolve during the design – but they are integral to UML. Table 4 shows the

Table 3. Typical Software Groupings

Title	Description
Package	A collection of logically grouped software modules. Examples: Built-In Test (BIT), Interface Drivers, Operational Flight Program
Module	Typically a collection of software classes. Examples: Communications BIT, Memory BIT, Serial Drivers, OFP Navigation
Class	Typically a collection of software methods. Examples: BIT External Interface, ARINC drivers, Air Data Inertial Reference Interface
Method	Typically a single-function software unit called by another method to perform some specific low-level service. Examples: Save Error Report, Service Interrupt, Compute Velocity, Format Serial Data. Code developers implement methods.

Table 4. Design Phases and Related UML Representations

Design Phase	UML Diagram	What Diagram Represents	Related Software Groupings
Requirements Analysis	Use Case	Operational requirements in logical groupings	
	Constraints	Performance requirements	
Requirements Allocation	Sequence, Collaboration	Interactions necessary to meet use case requirements	Class, Method (without details)
Architectural Design	Architectural	Software allocations in logical pieces	Package, Module, Class
Detailed Design	Data Flow	Data exchange among logical pieces	Class, Method (without details)
	Class	Related functions allocated in logical, encapsulated groups	Class, Method (with details)
Implementation, Test	none	n/a	Code

(imperfect) correspondence between the software design phases, UML diagrams, and software groupings. Table 4 is the basis for exploring the merits of software FMEA at various stages of a software design.

6. *RELATIVE MERITS OF ANALYSIS AT DIFFERENT STAGES OF DEVELOPMENT*

Several decisions must be made before performing software FMEA when the software is under design. When should the analysis be performed? What level of analysis detail is appropriate? What sort of results should be expected for a given budget and schedule? Who decides these things? The idea of FMEA on computer software is far from a well-understood concept, and related contract requirements are usually vague. The practical result is that most decisions will be made (or guided) by the analyst responsible for the effort, while the software development manager, as budget watcher, will naturally limit the cost – and therefore the scope – of the effort. The initial activity is therefore often a tradeoff exercise between the hours available for doing the work and the usefulness of results achievable in the available time.

6.1. *Method-level Analysis.*

FMEA at the code level requires the most extensive effort but gives the most useful results because it will identify more potential problems. This paper precludes code-level analysis (explored in Ref. 3). The next higher level (Table 3) is the method level. At this level, each *method* in each class is assumed to be given unexpected inputs – this is equivalent to failure of the method supplying the data or control, or receipt of bad data or control from the outside world. The analysis seeks to determine the local and system-level effects, and possibly intermediate-level effects, of each type of input failure.

Analysis at the method level requires detailed descriptions of each method's behavior, clear relationships between behavior and its inputs and outputs, relationships and data exchanges with other methods, and a complete description of system behavior. Since a FMEA is based on failure modes, the analyst must determine how each method can fail – and while a great deal of the conclusions will be intelligent guesswork, they are still guesswork. In addition, method-level FMEA may not be practical because methods may not be sufficiently described at the time of FMEA baseline definition. Worse, not all methods that eventually wind up in the design may have been identified at baseline time, and others may be dropped. In other words, a software design and its documentation must be pretty far along for a FMEA at the method level to be meaningful.

Despite these problems, FMEA at the method level may be acceptable – given the understanding that the analysis is based on a preliminary design. The conclusions can still be useful to software designers who must design their code to be sufficiently robust to detect unexpected inputs without causing system failures. Finally, due to the relatively large number of methods in a large project, analysis on this level can be the most costly to perform.

6.2. *Class-level Analysis.*

This could also be called object-level analysis. At the next level above the method level, FMEA performed at the class level can take less time because there are generally far fewer classes then methods, but compared to method analysis, class analysis must deal with increased ambiguity and, unless class behavior is well-defined, the analyst will need to supply intelligent guesswork more often. Your understanding of a class is limited by the class description, and perhaps descriptions the class's methods and attributes. A class-level analysis by definition should not include study of step-by-step PDL characterizations, and analysis of method details should

be declared out of scope. In fact, a full set of PDL characterizations – or any PDLs at all – may not exist if a FMEA must be started before design details have settled.

For class-level FMEA to be meaningful, the designers must have grouped functions into classes in a meaningful way. As a design goal, a software class should represent or be associated with one real-world thing – for example, an operator display capability. Unfortunately, classes sometimes include functionality not directly associated with one real-world thing, and in many designs some classes are designed simply to help other classes do their jobs – for example, data conversion classes. When you consider how a particular real-world class would behave with unexpected inputs, there may be dozens of failure effects – the nature of which requires some intelligent guesswork. Some failure effects may be very unclear because class behavior isn't clearly defined. Accordingly, class-level FMEA may be just an exercise to meet contract requirements. On the other hand, if class behavior is limited and well-defined, if relationships among classes and externals are clear, and if helper class functionality is lumped with the classes they assist, then an analyst should be able to conclude that bad input data will cause high-level effects that could be described as "inability to control horizontal stabilizer," "loss of error logging capability," or "loss of operator display." It must be understood that a class-level analysis is a high-level analysis that identifies only high-level failure effects.

6.3. *Module-level Analysis*

At the next level above classes, FMEA could be performed at the module level. A module of source code is usually a stand-alone software file that contains any number of classes in one long text file. FMEA at this level may not be worthwhile because too much detail must be ignored. The effort can also be viewed as giving obvious results. For example, if a module contains a set of classes that handle ARINC serial communications, then one can conclude that failure of this module to handle bad input data will result in loss of all ARINC data, and the FMEA report can simply list all hardware items that depend on ARINC data via this module. A FMEA at this level will not be nearly as extensive as FMEA at lower levels.

FMEA at module level may still be acceptable for large systems with extensive backup facilities or systems with many modules that individually contain relatively few classes. The more modules in a system, the more meaningful the FMEA.

6.4. *Package-level analysis*

At the highest level, FMEA probably makes the least sense because of the obvious and limited nature of the results. For example, loss of a Built-In Test package or loss of a Hydraulics package is a system-level consideration that is arguably beyond the scope of a software analysis. FMEA at this level is unlikely to meet analysis requirements. Even so, analysis of this sort is a good start to an analysis at a lower level because many lower-level failures will ultimately yield failure effects at the package level.

7. *SUGGESTIONS FOR FMEA AT ANY SOFTWARE LEVEL*

It is a good idea to for the analysis team to develop rules and guidelines before starting the FMEA (Ref. 4). During the effort, assumptions about software behavior should be worst-case, and expect the software designers to take issue with your conclusions. A database tool (Ref. 3) will speed the FMEA effort, minimize human errors, and make it much easier to produce and revise the final report.

REFERENCES

1. Douglass, Bruce Powel, *Real-Time UML, Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd Edition,* Addison-Wesley 2000, ISBN 0201657848

2. Haapanen Pentti, Helminen Atte, "Failure Mode and Effects Analysis of Software-Based Automation Systems," STUK-YTO-TR 190, Helsinki 2002. Available at www.stuk.fi/julkaisut/tr/stuk-yto-tr190.html.

3. Ozarin, Nathaniel and Siracusa, Michael., "A Process for Failure Modes and Effects Analysis of Computer Software," *Proceedings of the Annual Reliability and Maintainability Symposium*, January 2003.

4. Ozarin, Nathaniel, "Developing Rules for Failure Modes and Effects Analysis of Computer Software, Proceedings of SAE Conference on Aircraft Safety, September 2003.

BIOGRAPHY

Nathaniel W. Ozarin
The Omnicon Group Inc.
40 Arkay Drive
Hauppauge, NY 11788 USA

nozarin@omnicongroup.com

Nat Ozarin is a senior engineering consultant at The Omnicon Group Inc., a company specializing in reliability and safety analysis for the military, medical, industrial, and transportation industries. His background includes hardware engineering, software engineering, systems engineering, programming, and reliability engineering. He received a BSEE from Lehigh University, an MSEE from Polytechnic University of New York, and an MBA from Long Island University. He is an IEEE member.