Trends in Embedded Software Engineering

Peter Liggesmeyer, *University of Kaiserslautern*

Mario Trapp, *Fraunhofer Institute for Experimental Software Engineering*

The increasing complexity of functional and extrafunctional requirements for embedded systems calls for new software development approaches.

Over the last 20 years, software's impact on embedded system functionality, as well as on the innovation and differentiation potential of new products, has grown rapidly. This has led to an enormous increase in software complexity, shorter innovation cycle times, and an ever-growing demand for extrafunctional requirements—software safety, reliability, and timeliness, for example—at affordable costs.

Embedded software development has long been reduced to a purely laborious task, necessary to implement functionality such as control algorithms with programming languages. With embedded software's increasing complexity and quality demands, however, software engineering's role is becoming more important.

Software development is shifting from manual programming to model-driven development (MDD). Here, we identify specific characteristics of embedded software, describe the paradigm of model-driven software engineering, and describe challenges and solutions to assure the quality of embedded software.

Embedded Software

Developers of modern IT systems use standardized platforms that provide the basic infrastructure for service-oriented architectures, distributed systems, error recovery, and reliability. Several abstraction layers and common platforms support platform-independent and distributed systems. Virtualization techniques provide scalability through the dynamic scaling of (virtual) hardware platforms, and live migration and hot-spare hardware increase IT system reliability.

However, such an approach doesn't fit the embedded systems domain. Resource requirements originating from cost, energy, size, or weight constraints demand the efficient use of available hardware resources. So, heavyweight abstraction layers, platforms, and virtualization techniques that require many resources for mapping high-level platforms to concrete hardware devices aren't feasible.

The diversity of embedded systems also prevents the creation of a single specialized platform. Embedded systems take such various forms as mobile phones, train-control systems based on programmable logic controllers (PLCs), and glucose meters. Obviously, no single embedded system exists. Moreover, embedded software is rarely a stand-alone product; rather, it's a single element in a product consisting of mechanics, electrics and electronics, and software. Embedded system development always focuses on the product, so it must consider various constraints. For one, many embedded systems are mass-produced products

With the rapidly growing complexity of embedded software systems, many companies have run into software quality problems.

with tight cost restrictions. Moreover, embedded systems developers must consider resource limitations and extreme environmental conditions such as heat, humidity, or radiation.

To meet these requirements, developers often create a product-specific hardware platform. Consequently, embedded software must be tailored to heterogeneous hardware platforms and operating systems to meet high-performance demands despite limited resources (such as memory or processing power). This yields tailored, specific software platforms that are either created through the application and adaptation of platform frameworks or developed specifically for one embedded system. The need to create specialized hardware and software platforms is a key difference between IT systems and embedded systems. For example, developers' efforts in creating the Automotive Open System Architecture (www.autosar.org) development platform—a common standardized platform for developing embedded systems in automobiles—illustrate the difficulties in establishing a common platform for even one domain in embedded system development.

Extrafunctional requirements contribute to these difficulties. Such requirements are far more important in embedded systems than in IT systems. For example, IT system users are accustomed to waiting for the system to react, but a car won't stop moving to wait for its control system to return a calculation. Domain-specific, extrafunctional requirements, such as software safety, reliability, and timeliness, must be integrated into the development process. Safety-critical systems, for example, need certificated development processes and tool chains according to standards such as International Electrotechnical Commission (IEC) 61508.[1] Fulfilling rigorous extrafunctional properties is therefore costly and time consuming. The regular need to adapt hardware, platforms, and operating systems also limits generic solutions' applicability. Moreover, resource, cost, and other limitations often conflict with extrafunctional requirements (such as real-time requirements).

In addition, embedded systems developers require extensive domain knowledge. For example, developing a vehicle stability control system is impossible if you don't understand the physics of vehicle dynamics. Consequently, embedded software development has been restricted mainly to control engineers and mechanical engineers, who have the necessary domain expertise. With the rapidly growing complexity of embedded software systems, however, many companies have run into software quality problems. Supporting seamless cooperation between domain and software development experts to combine their complementary expertise remains a core challenge in embedded software development.

From Programming to Model-Driven Engineering

Managing the rapidly increasing complexity of embedded software development is one of the most important challenges for increasing product quality, reducing time to market, and reducing development cost. MDD is one of the promising approaches that have emerged over the last decade. Instead of directly coding software using programming languages, developers model software systems using intuitive, more expressive, graphical notations, which provide a higher level of abstraction than native programming languages. In this approach, generators automatically create the code implementing the system functionalities. To manage embedded systems' growing complexity, modeling will likely replace manual coding for application-level development—just as high-level programming languages have almost completely replaced assembly language.

Model-driven architecture (MDA; www.omg.org/mda) is the primary driver for MDD of IT systems. Although MDA defines generic concepts of models and model transformations, MDD of embedded systems started much earlier, before UML and MDA were standardized. Therefore, its main drivers have been modeling tools implementing vendor-specific languages (for example, Matlab/Simulink and LabView). Using these tools, developers can completely specify embedded software systems using high-level models. Researchers have devised many approaches focusing on the specific aspects of model-driven embedded systems development.[2] In addition to industry approaches are those representing ongoing research—for example, formal verification of hybrid systems,[3] modeling of hybrid systems with UML,[4] or modeling the requirements of hybrid systems.[5]

MDD of embedded systems is, as with any other development activity, controlled by a development process. Figure 1 illustrates a possible iterative MDD process for the embedded systems domain that covers the complete software development life cycle. The literature and development standards have proposed similar variants. These variants include the life cycles defined in standards such as the IEC 61508;[1] the UML-based Ropes (Rapid Object-Oriented Process for Embedded Systems),[6] which is based on the spiral process model;[7] and different approaches based on a V-Model.[8]

Requirements-engineering tools support requirements specification, manage requirement changes, and help track test case results and test coverage. Depending on the modeling languages used to describe requirements, developers can also use graphical languages such as UML or the Systems Modeling Language (SysML) to model requirements—for example, using requirements diagrams or use cases and scenarios. Developers can create a system's functional design based on the system requirements. Functional designs cover an embedded system's functionalities without considering any technical implementation details. Instead, they consist of a computationally independent model—for example, a mathematical model describing the created software system's core behavior. For example, a functional design would specify an automotive system's closed-loop controllers from a control engineer's viewpoint, omitting implementation details such as tasking or deployment. Not all embedded software systems require a functional design, so developers might skip this step.

Developers define the actual software architecture on the basis of the requirements and functional design. Because defining software architecture is a creative process, transforming a functional design to an architecture is a mostly manual process. An architecture definition consists of various views covering the architecture's different aspects.[9] Because UML supports different diagrams and is easily extendable, developers often use it to specify architectures, including those of embedded systems. Few data-flow-oriented modeling tools provide all required language concepts and flexibility, so they're rarely applicable to architectural modeling.

After defining the system architecture, developers refine the different components and connections identified during the architecture specification to obtain an executable, but still platform-independent, system. To this end, they identify further subcomponents and model the basic components' actual behavior. One solution is to use UML to model the refined structure and data-flow languages to model component behavior—particularly if they need to model continuous or hybrid behavior such as (closed-loop) controllers or signal-processing components. UML supports data-flow-oriented models to define the structure. Developers can use additional diagrams—for example, sequence diagrams—to specify the interaction between components. Special deployment and tasking diagrams support the modeling of multithreaded and distributed applications.

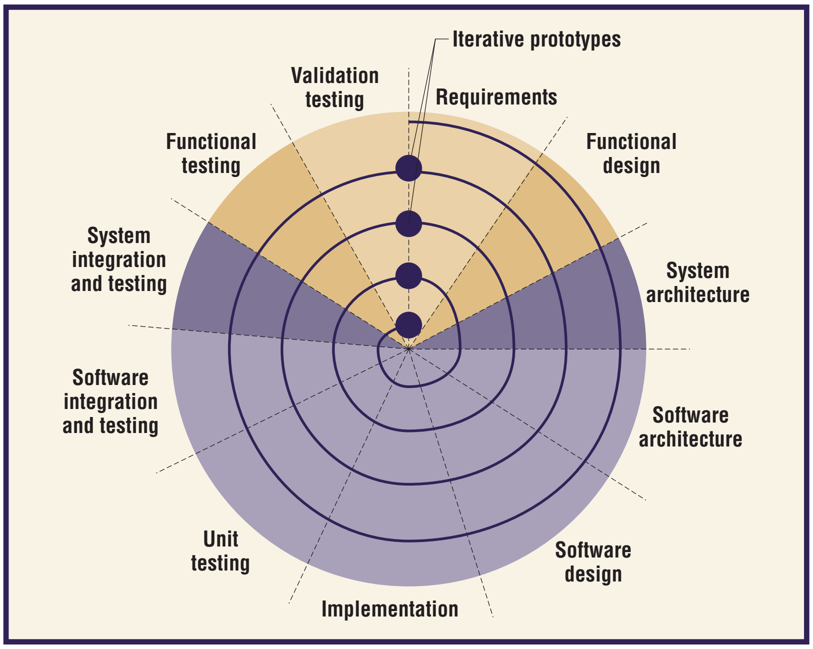


Figure 1. An iterative model-driven development process for an embedded system would encompass all phases of the development life cycle. This process lets developers produce partial implementations after each iteration, with the most critical aspects implemented first.

In principle, these platform-independent models are sufficient for generating executable code. Usually, however, developers must further refine or extend platform-independent models in the platform-specific design to support efficient code generation for the target execution platform.

An iterative software development process such as that in Figure 1 supports partial implementations, with the system's most critical aspects implemented first, as is often imperative in the embedded systems domain. Early iterations and integrations let developers react quickly to necessary changes originating, for instance, from resource limitations. Today, a lack of tool support and integration makes it impossible to seamlessly cover the complete development life cycle using model-driven-engineering paradigms. Particularly for the design phase, however, automated transformation from design models to code is already possible. This allows rapid creation of executable prototypes as well as early evaluation of conformance to extrafunctional requirements, such as resource use and timeliness.

Following the shift from assembly language to high-level programming languages, and from programming to model-based design, comes the shift from MDD to domain-specific development[10]—a shift already on the horizon. Some industry case

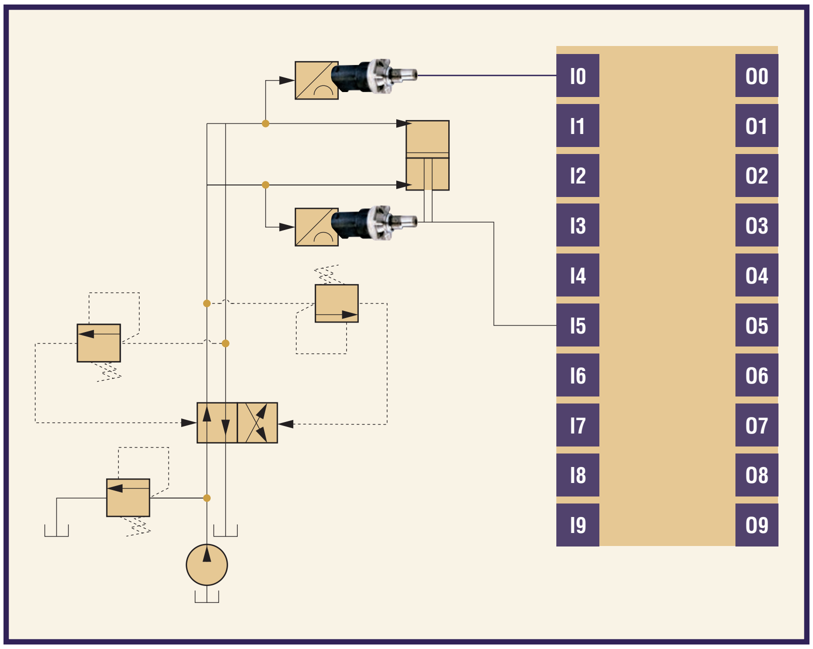


Figure 2. Example of a domain-specific language. In this view of the model, developers simply define which sensors they use to monitor the hydraulics system and connect them to the respective input ports of the monitoring hardware. The complete code to configure the sensors and to retrieve and to interpret the sensor readings is then generated completely automatically.

studies have already successfully applied domain-specific development. MDD is based on general-purpose languages and code generators for different application domains. This claim for generality contradicts the optimization of the language and code generators to a given application context, and to the hardware platform used. Domain-specific modeling lets developers use domain-specific concepts, so it provides even more-intuitive modeling languages and integrates more software developer know-how and application-specific optimizations into the code generators.

Figure 2 shows a domain-specific language for specifying condition-monitoring systems for hydraulic systems. Although the example looks like a domain expert's Visio drawing, it's actually a formal model with a clear syntax and semantics, enabling code generation based on this specification. Because the modeling elements have rich semantics and the family of possible target platforms is known, the resulting tailored code generators produce efficient code. All the software development expertise necessary to create a system from this specification is encapsulated in the code generator, libraries, and (if required) the platform software, which might include an operating system or a microkernel. Here, software engineering experts are needed primarily to create the domain-specific language and tool chain, which domain experts can then use to build an embedded software system.

Although domain-specific modeling simplifies system specification, the modeling language is limited to a specific class of problems. So, the development of the domain-specific language and generators requires an initial effort, which will only pay off if the same development environment is applicable to several projects. In certain application domains, such as the condition-monitoring system example, this is certainly the case. For these domains, domain-specific development is an efficient, promising approach. In general, however, domain-specific development won't completely replace MDD; rather, hybrid solutions are more likely.

For example, in UML, profiles let developers tailor and extend the generic language to specific domains, thereby specializing the language to the domain and enriching its semantics so that it can be used more intuitively. Nonetheless, because UML is still at its base, the specialized language can reuse existing (code generation) tool chains. UML's complete expressiveness is available if domain-specific extensions are insufficient. Next-generation languages and tools will provide extended extension and tailoring mechanisms, potentially combining the advantages of MDD's generality with those of specialized domain-specific modeling languages.

Quality Assurance of Safety-Related Systems

In addition to the constructive phases of embedded systems development, ensuring the quality of a system's functional and extrafunctional properties is crucial in such development. This is particularly true for safety-related systems.

Before code generation, developers can apply static and formal verification techniques to ensure software quality; after code generation, they can use dynamic-testing approaches. Regarding quality assurance, dynamic testing is still a widespread approach for determining the correct functioning of software and systems. However, only formal verification techniques can achieve complete correctness proofs. Still, every formal technique has disadvantages. Applying formal techniques to a complete, complex software system isn't possible. Moreover, formally proven software might still be unsafe, and safe software might not be completely correct. Correctness clearly supports safety, but it doesn't substitute for safety analysis.

Safety analysis is therefore another central element in the development of safety-related systems, as well as for software. Model-based or measurement-based approaches—each of which has advantages and disadvantages—can address safety and reliability issues. In practice, a combination of approaches is often necessary.

Model-based safety and reliability analysis provides information at an early stage—that is, before the system is implemented. It might identify problems early and reduce rework costs. However, developing the appropriate models requires additional effort and time, because safety models such as fault trees[11] are usually generated manually.

The measurement-based approach to safety and reliability is only applicable after the system has been fully specified or implemented. Because measurements—such as failure data from system testing—are never complete, producing dependable results requires statistical techniques.

Dynamic Testing

Although formal verification techniques seem to fit safety-critical software requirements, many developers use dynamic testing—that is, testing by executing defined test cases. Particularly in the MDD context, when using iterative development processes, developers can apply tests based on the design models in early design phases, enabling a continuous quality-assurance process. Unlike formal techniques, dynamic tests can be applied to complex systems and are easier to realize. Basically, testing can start in the first iteration of the development process, directly after code is generated from the software design. As Figure 1 shows, you can apply dynamic testing in multiple phases:

- *Unit testing* tests specific units.
- *Software and system integration testing* tests the integration of software units and of the software system with its hardware.
- *Validation testing* ensures that all system requirements have been fulfilled.
- Optional *functional testing* ensures conformance to the functional design.

Although tests can't prove software correctness, systematic testing can achieve sufficient software quality—even in safety-critical applications. Combining statistic dynamic-testing procedures with statistic analyses based on reliability growth models allows quantification of residual risks.

Developers can implement testing in the actual operating environment, which lets them detect certain problems, such as those usually not detected through formal verification. These might include interface problems but could also be the violation of extrafunctional properties resulting in timing, communication, or resource-consumption problems originating from the integration of software and hardware.

Testing is definitely the simplest way to evaluate system behavior, even after deployment, and with respect to extrafunctional properties. In addition, dynamic testing is scalable. A thorough, systematic selection of test cases ensures that they appropriately cover the desired functionality. Accepted minimal criteria for testing embedded software are the complete coverage of the specification with test cases, branch coverage of the code, and reproducibility of the test (regression testing). Such criteria are augmented with additional requirements in special cases—for example, for safety-critical software for commercial aircraft (RTCA DO-178B[12]).

Model-Based Safety and Reliability Analysis

Despite the importance of standard quality-assurance techniques such as testing and formal verification for embedded software development, they can't replace safety analysis techniques. So, the demand is strong for techniques and models that help developers achieve and assess safety and reliability. These techniques include reliability block diagrams, fault tree analysis (FTA),[11] and Markov models.

FTA depicts causal chains leading to a failure as a tree. The system failure to be examined is the tree's root; the basic failures—component failures, for example—are its leaves. This tree structure inherently describes a hierarchical breakdown, but with respect to the hierarchy of failure influences rather than the system architecture. In FTA, modules are subtrees. This partitioning merely represents a property of the influence chains. The modules generally don't correspond to the software components identified during system development. In a software model, the connections of components usually define a graph structure that can't be mapped directly to fault trees. Consequently, division of labor is hampered during fault tree design. Moreover, tracing fault tree elements to the corresponding software components is difficult. Reusing software components makes it difficult to reuse the corresponding fault tree elements, as well.

To overcome these drawbacks, component fault trees[13] extend conventional fault trees using real components that are connected via ports (see Figure 3 on the next page). The fault tree structure can therefore be similar to the software system's structure, simplifying the definition of the

Although tests can't prove software correctness, systematic testing can achieve sufficient software quality.

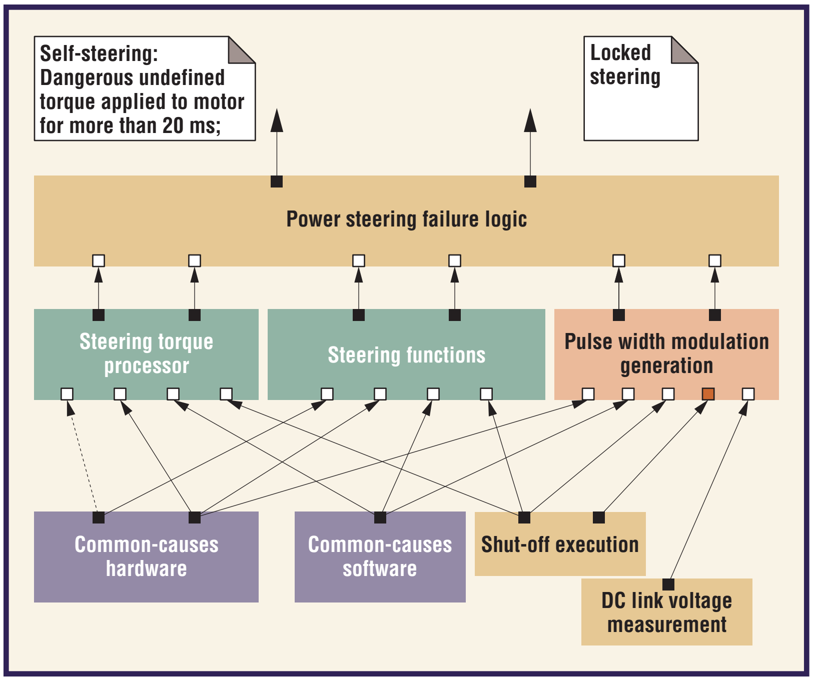

Self-steering: Dangerous undefined torque applied to motor for more than 20 ms;
Locked steering
Power steering failure logic
Steering torque processor
Steering functions
Pulse width modulation generation
Common-causes hardware
Common-causes software
Shut-off execution
DC link voltage measurement

Figure 3. Component fault trees simplify fault tree analyses of complex software systems (www.essarel.de). The fault tree describes two top events of an active power steering system—namely, locked steering and self-steering. The active power steering system is decomposed to different components such as the power steering torque processor. The component fault tree follows the structure of the system and thus simplifies the mapping between functional model and safety model, enables division of labor, and improves the reusability of fault tree components.

safety model and the traceability to the design models. By providing a real-component concept, component fault trees directly support the division of labor and intensive component reuse.

Measurement-Based Safety and Reliability Analysis

Safety engineers can use the modeling techniques we've described in the early design phases to constructively guide the development process, and thus prevent safety-critical design faults. However, the result's accuracy depends on the safety and reliability models' quality. Measurement-based approaches don't suffer from modeling faults. In addition, the validity of measurements on a real entity is unquestionable. However, like dynamic testing, these approaches are applicable after code generation at the earliest. Moreover, measurements are never complete. Applying statistical methods to obtain a statistically protected trustworthiness can somewhat compensate for this disadvantage. Fault trees let you model reliability; reliability growth models let you measure, evaluate, and predict reliability. An overview of software reliability modeling—a research discipline since the early 1970s—is available elsewhere.[14]

Experience shows that the application of theory is critical in software reliability analysis. Reliability must be easy to measure and predict, without the user having to completely understand its theoretical background. Almost all the theories can be encapsulated in a tool that evaluates whether a certain reliability growth model fits the observed failure data, determines model parameters, and calculates reliability values (for example, failure counts and rates).

Up to now, researchers have considered safety and reliability separately from model-driven engineering. Even emerging standards such as ISO/CD 26262 for the automotive domain insufficiently address model-driven engineering of embedded software. A challenge is therefore to systematically define how to ensure safety in the model-driven-engineering context—and not only through verification and validation.

We must also clarify how different demands will be applied to MDD approaches. MDD concepts could be beneficial in safety engineering. Particularly for engineering safety-critical systems, isolated concepts available in research must evolve into standard approaches accepted by certification bodies.

Different approaches for the development and quality assurance of embedded software systems have been successfully transferred to industry. However, techniques for systematically developing embedded software can hardly keep up with the ever-growing demand for new functionalities and technologies. In the development of safety-critical systems, new functionalities are useless if you can't ensure their quality and safety. The rapid progress of systematic software engineering technologies will therefore be a key factor in the successful future development of even more-complex embedded systems.

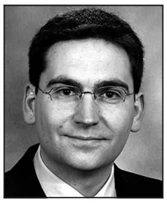
Acknowledgments
We thank Thomas Kuhn for his comments and contribution to this article. We also thank the reviewers and editors for their comments, which helped us improve this article's quality.

References
1. *IEC 61508, Functional Safety of Electrical/Electronical/Programmable Electronic Safety-Related Systems*, Int'l Electrotechnical Commission, 1998.

2. H. Giese and S. Henkler, "A Survey of Approaches for the Visual Model-Driven Development of Next Generation Software-Intensive Systems," *J. Visual Languages and Computing*, vol. 17, no. 6, 2006, pp. 528–550.
3. R. Alur et al., "Hierarchical Hybrid Modeling of Embedded Systems," *Proc. 1st Int'l Workshop Embedded Software* (EMSOFT 01), LNCS 2211, Springer, 2001, pp. 14–31.
4. K. Berkenkötter et al., "Executable HybridUML and Its Application to Train Control Systems," *Integration of Software Specification Techniques for Applications in Eng.*, LNCS 3147, Springer, 2004, pp. 145–173.
5. R. Grosu, I. Krüger, and T. Stauner, "Hybrid Sequence Charts," *Proc. 3rd IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC 00), IEEE Press, 2000, p. 104.
6. B.P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems,* Addison-Wesley, 2004.
7. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, no. 5, 1988, pp. 61–72.
8. B.W. Boehm, "Guidelines for Verifying and Validating Software Requirements and Design Specification," *Proc. European Conf. Applied Information Technology* (Euro IFIP), North-Holland, 1979.
9. P. Kruchten, "The 4 + 1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 42–50.
10. S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley-IEEE CS Press, 2008.
11. *IEC 61025, Fault Tree Analysis (FTA)*, Int'l Electrotechnical Commission, 1990.
12. *RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics, 1992.
13. B. Kaiser, P. Liggesmeyer, and O. Mäckel, "A New Component Concept for Fault Trees," *Proc. 8th Australian Workshop Safety Critical Systems and Software* (SCS 03), Australian Computer Soc., 2003, pp. 37–46.
14. M.R. Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill, 1995.

About the Authors

Peter Liggesmeyer is a full professor at the University of Kaiserslautern and the director of the Fraunhofer Institute for Experimental Software Engineering . His research interests include engineering of dependable systems, software quality assurance, and software visualization. Liggesmeyer has a doctorate in electrical engineering from the University of Bochum. He's a member of the IEEE and the German Chapter of the ACM. Contact him at peter.liggesmeyer@iese.fraunhofer.de.



Mario Trapp is a division manager at the Fraunhofer Institute for Experimental Software Engineering. His research interests include model-driven development of high-integrity embedded systems and safety-engineering techniques. Trapp has a doctorate in computer science from the University of Kaiserslautern. Contact him at mario.trapp@iese.fraunhofer.de.