Embedded Software Engineering: The State of the Practice

Bas Graaf, Marco Lormans, and Hans Toetenel, *Delft University of Technology*

Many products today contain software (for example, mobile telephones, DVD players, cars, airplanes, and medical systems). Because of advancements in information and communication technology, in the future even more products will likely contain software. The market for these "enhanced" products is forecasted to grow exponentially in the next 10 years.[1] Moreover, these embedded systems' complexity is increasing, and the amount and variety of software in these

products are growing. This creates a big challenge for embedded-software development. In the years to come, the key to success will be the ability to successfully develop high-quality embedded systems and software on time. As the complexity, number, and diversity of applications increase, more and more companies are having trouble achieving sufficient product quality and timely delivery. To optimize the timeliness, productivity, and quality of embedded software development, companies must apply software engineering technologies that are appropriate for specific situations.

Unfortunately, the many available software development technologies don't take into account the specific needs of embedded-systems development. This development is fundamentally different from that of nonembedded systems. Technologies for the development of embedded systems should address specific constraints such as hard timing constraints, limited memory and power use, predefined hardware platform technology, and hardware costs. Existing development technologies don't address their specific impact on, or necessary customization for, the embedded domain. Nor do these technologies give developers any indication of how to apply them to specific areas in this domain—for example, automotive systems, telecommunications, or consumer electronics. Consequently, tailoring a technology for a specific use is difficult. Furthermore,

An inventory of eight European companies reveals what tools developers of embedded-systems software are and aren't using, and why. The need exists for more specific, yet flexible, tools.

A Sample of the Interview Outline

Here are some discussion topics and questions from the outline we used for the interviews.

Technology

What are the most important reasons for selecting development technologies?
Impact of introducing new technologies (cost, time, and so on).
Why not use modern/different technologies?

Software life cycle

Software requirements engineering

How are the requirements being gathered?

- What are the different activities?
- What documents are produced?
- What about tool support?

How are the requirements being specified?

- What specification language?
- What about tool support? (Consider cost, complexity, automation, training, acceptance)
- What notations/diagrams?
- What documents are produced?
- How are documents reviewed?
- What are advantages/disadvantages of followed approaches?

Software architecture design

How is the architecture specified?

- What architecture description language?
- What about tool support? (Consider cost, complexity, automation, training, acceptance)
- Are design patterns used?
- What notations/diagrams?
- What documents are produced?
- How are documents reviewed?
- What are advantages/disadvantages of followed approaches?

the embedded domain is driven by reliability factors, cost factors, and time to market. So, this embedded domain needs specifically targeted development technologies.

In industry, the general feeling is that the current practice of embedded software development is unsatisfactory. However, changes to the development process must be gradual; a direction must be supplied. To achieve this, we need more insight into the currently *available* and currently *used* methods, tools, and techniques in industry.

To gain such insight, we performed an industrial inventory as part of the MOOSE (Software Engineering *Meth*odologies for *E*mbedded Systems, www.mooseproject.org) project. MOOSE is an ITEA (Information Technology for European Advancement, www.itea-office.org) project aimed at improving software quality and development productivity for embedded systems. Not only did we gain an overview of which technologies the MOOSE consortium's industrial partners use, we also learned why they use or don't use certain technologies. In addition, we gained insight into what currently unavailable technologies might be helpful in the future.

Methods and scope

The inventory involved seven industrial companies and one research institute in three European countries (see Table 1). These companies build a variety of embedded software products, ranging from consumer electronics to highly specialized industrial machines. We performed 36 one-hour interviews with software practitioners. The respondents were engineers, researchers, software or system architects, and managers, with varying backgrounds. To get a fair overview of the companies involved (most of which are very large), we interviewed at least three respondents at the smaller companies and five or six at the larger companies.

We based the interviews on an outline specifying the discussion topics (see the sidebar). To be as complete as possible, we based this outline on a reference process model. Because software process improvement methods have such a (ideal) process model as their core, we used one of them. We chose the BOOTSTRAP methodology's (www.bootstrap-institute.com) process model because of its relative emphasis on engineering processes compared to other process models, such as those of the Capability

Maturity Model and SPICE (*Software Process Improvement and Capability Determination*).[2] BOOTSTRAP's other advantage for this inventory is that the BOOTSTRAP Institute developed it with the European software industry in mind.

For every interview we created a report; we consolidated the reports for a company into one report. We then analyzed the company reports for trends and common practices. Finally, we wrote a comprehensive report that, for confidentiality reasons, is available only to MOOSE consortium members. That report forms the basis for this discussion.

Results

When considering the embedded software development process, you need to understand the context in which it is applied. After all, most companies that develop embedded software do not sell it. They sell mobile phones, CD players, lithography systems, and other products. The software in these products constitutes only one (important) part. Embedded-software engineering and other processes such as mechanical engineering and electrical engineering are in fact subprocesses of systems engineering. Coordinating these subprocesses to develop quality products is one of embedded-system development's most challenging aspects. The increasing complexity of systems makes it impossible to consider these disciplines in isolation.

For instance, when looking at communication between different development teams, we noticed that besides vertical communication links along the lines of the hierarchy of architectures, horizontal communication links existed. Vertical communication occurs between developers who are responsible for systems, subsystems, or components at different abstraction levels (for example, a system architect communicating with a software architect). Horizontal communication occurs between developers who are responsible for these things at the same abstraction level (for example, a programmer responsible for component A communicating with a programmer responsible for component B).

Still, we found that systems engineering was mostly hardware driven—that is, from a mechanical or an electronic viewpoint. In some companies, software architects weren't even involved in design decisions at the system level. Hardware development primarily dominated system development because of longer lead times and logistical dependencies on external suppliers. Consequently, software development started when hardware development was already at a stage where changes would be expensive. Hardware properties then narrowed the solution space for software development. This resulted in situations where software inappropriately fulfilled the role of integrator; that is, problems that should have been solved in the hardware domain were solved in the software domain. Embedded-software developers felt that this was becoming a serious problem. So, in many companies this was changing; software architects were becoming more involved on the system level.

Depending on the product's complexity, projects used system requirements to design a system architecture containing multidisciplinary or monodisciplinary subsystems. (A multidisciplinary subsystem will be implemented by different disciplines; a discipline refers to software, or mechanics, or electronics, or optics, and so on. A monodisciplinary subsystem will be implemented by one discipline.) Next, the projects allocated system requirements to the architecture's different elements and refined the requirements. This process repeated for each subsystem. Finally, the projects decomposed the subsystems into monodisciplinary components that an individual developer or small groups of developers could develop. The level of detail at which decomposition resulted in monodisciplinary subsystems varied. In some cases, the first design or decomposition step immediately resulted in monodisciplinary subsystems and the corresponding requirements. In other cases, subsystems remained multidisciplinary for several design steps.

This generic embedded-systems-development process resulted in a tree of requirements and design documents (see Figure 1). Each level represented the system at a specific abstraction level. The more complex the system, the more evident this concept of abstraction levels was in the development process and its resulting artifacts (for example, requirements documentation).

In the process in Figure 1, requirements on different abstraction levels are related to each other by design decisions, which were recorded in architecture and design documentation. At the system level, these decisions concerned par-

> We found that systems engineering was mostly hardware driven— that is, from a mechanical or an electronic viewpoint.
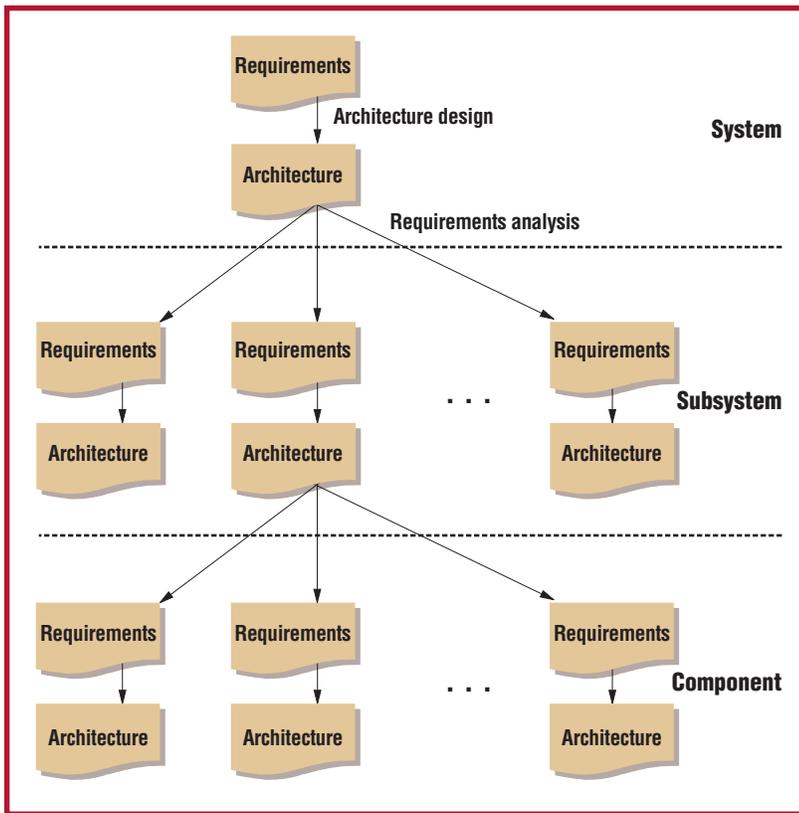
Figure 1. The decomposition of the embedded-systems-development process.

titioning of the functional and nonfunctional requirements over software and hardware components. The criteria used for such concurrent design (codesign) were mostly implicit and based on system architects' experience.

Requirements engineering

Typically, embedded-systems development involved many stakeholders. This was most apparent during requirements engineering. Figure 2 depicts our view of the most common stakeholders and other factors.

Figure 2. Embedded-systems-development stakeholders and other factors.
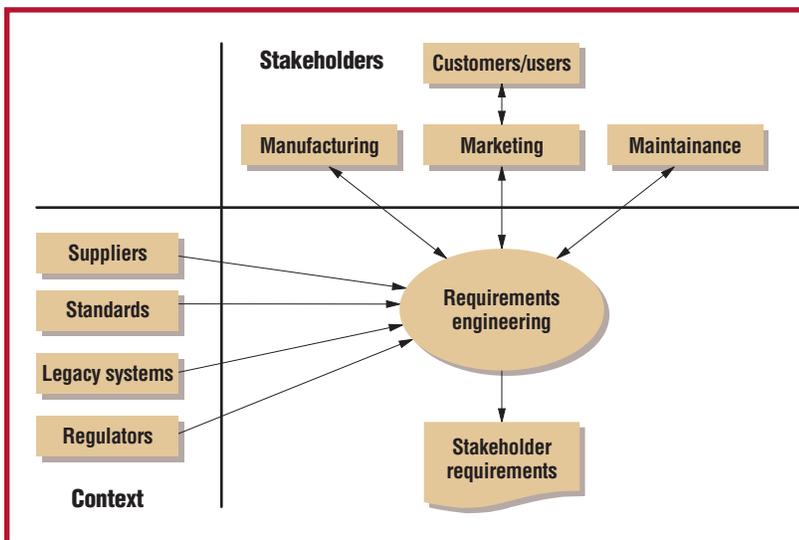
In requirements engineering's first phase, the customer determines the functional and nonfunctional requirements. Depending on the product domain, the customer negotiates the requirements via the marketing and sales area or directly with the developers.

The first phase's output is the agreed requirements specification, which is a description of the system that all stakeholders can understand. This document serves as a contract between the stakeholders and developers. At this point, we noticed a clear difference between small and large projects. In small projects, the stakeholder requirements also served as developer requirements. In large projects, stakeholder requirements were translated into technically oriented developer requirements.

Requirements specify what a system does; a design describes how to realize a system. Software engineering textbooks strictly separate the requirements and the design phases of software development; in practice, this separation is less obvious. In fact, the small companies often put both the requirements and design into the system specification. These companies did not translate the system requirements explicitly into software requirements. The development processes in the larger companies did result in separate requirements and design documents on different abstraction levels. However, in many cases, these companies directly copied information from a design document into a requirements document for the next abstraction level instead of first performing additional requirements analysis.

Requirements specification. Companies usually specified requirements in natural language and processed them with an ordinary word processor. They normally used templates and guidelines to structure the documents. The templates prescribed what aspects had to be specified. However, not all projects at a company used these templates, so requirements specifications from different projects sometimes looked quite different.

Because embedded software's nonfunctional properties are typically important, we expected these templates to reserve a section on nonfunctional requirements next to functional requirements. This wasn't always the case. For example, the requirements specification didn't always explicitly take into account

real-time requirements. Sometimes a project expressed them in a separate section in the requirements documents, but often they were implicit. Requirements specification and design also usually didn't explicitly address other typical embedded-software requirements, such as those for power consumption and memory use.

Projects that employed diagrams to support requirements used mostly free-form and box-line diagrams in a style that resembles the Unified Modeling Language, dataflow diagrams, or other notations. Project members primarily used general-purpose drawing tools to draw the diagrams. Because of the lack of proper syntax and semantics, other project members often misinterpreted the diagrams. This was especially true for project members working in other disciplines that employ a different type of notation.

UML was not common practice yet, but most companies were at least considering its possibilities for application in requirements engineering. Use cases were the most-used UML constructs in this phase. Some projects used sequence diagrams to realize use cases; others applied class diagrams for domain modeling. However, the interpretation of UML notations was not always agreed on during requirements engineering. It wasn't always clear, for instance, what objects and messages in UML diagrams denote when a sequence diagram specifies a use case realization.

On the lowest levels, projects commonly used pre- and postconditions to specify software requirements. They specified interfaces as pre- and postconditions in natural language, C, or some interface definition language.

Projects rarely used formal specifications. One reason was that formal specifications were considered difficult to use in complex industrial environments and require specialized skills. When projects did use them, communication between project members was difficult because most members did not completely understand them. This problem worsened as projects and customers needed to communicate. In one case, however, a project whose highest priority was safety used the formal notation Z for specification.

Requirements management. When looking at Figures 1 and 2, you can imagine that it's hard to manage the different requirements from all these different sources throughout develop-

ment. This issue was important especially in large projects.

Another complicating factor was that most projects didn't start from scratch. In most cases, companies built a new project on previous projects. So, these new projects reused requirements specifications (even for developing a new product line). Consequently, keeping requirements documents consistent was difficult. To keep all development products and documents consistent, the projects had to analyze the new features' impact precisely. However, the projects frequently didn't explicitly document relations between requirements, so impact analysis was quite difficult. This traceability is an essential aspect of requirements management. Tracing requirements was difficult because the relations (for example, between requirements and architectural components) were too complex to specify manually.

Available requirements management tools didn't seem to solve this problem, although tailored versions worked in some cases. A general shortcoming of these tools was that the relations between the requirements had no meaning. In particular, tool users could specify the relations but not the rationale behind the link.

When projects did document relations between requirements, they used separate spreadsheets. Some companies were using or experimenting with more advanced requirements management tools such as RequisitePro (Rational), RTM (Integrated Chipware), and DOORS (Telelogic). These experiments weren't always successful. In one case, the tool's users didn't have the right skills, and learning them took too long. Also, the tool handled only the more trivial relations between requirements, design, and test documents. So, developers couldn't rely on the tool completely, which is important when using a tool.

Requirements management also involves release management (managing features in releases), change management (backwards compatibility), and configuration management. Some requirements management tools supported these processes. However, because most companies already had other tools for this functionality, integration with those tools would have been preferable.

Architecture

Small projects didn't always consider the explicit development, specification, and analysis

UML was not common practice yet, but most companies were at least considering its possibilities for application in requirements engineering.

of the product architecture necessary. Also, owing to time-to-market pressure, the scheduled deadlines often obstructed the development of sound architectures. Architects often said they didn't have enough time to do things right.

The distinction between detailed design and architecture seemed somewhat arbitrary. During development, the projects interpreted architecture simply as high-level design. They didn't explicitly use other architectural aspects, such as nonlocality,[3] that are often considered in the scientific literature.

Architecture design. Designing a product's or subsystem's architecture was foremost a creative activity that was difficult to divide into small, easy-to-take steps. Just as system requirements formed the basis for system architecture decisions, system architecture decisions constrained the software architecture.

In some cases, a different organizational unit had designed the system architecture. So, the architecture was more or less fixed—for instance, when the hardware architecture was designed first or was already known. This led to suboptimal (software) architectures. Because software was considered more flexible and has a shorter lead time, projects used it to fix hardware architecture flaws, as we mentioned before.

The design process didn't always explicitly take into account performance requirements. In most cases where performance was an issue, projects just designed the system to be as fast as possible. They didn't establish how fast until an implementation was available. Projects that took performance requirements into account during design did so mostly through budgeting. For example, they frequently divided a high-level real-time constraint among several lower-level components. This division, however, often was based on the developers' experience rather than well-funded calculations. Projects also used this technique for other nonfunctional requirements such as for power and memory use.

Projects sometimes considered COTS (commercial off-the-shelf) components as black boxes in a design, specifying only the external interfaces. This was similar to an approach that incorporated hardware drivers into an object-oriented design. However, developers of hardware drivers typically don't use OO techniques. By considering these drivers as black boxes and looking only at their interfaces, the designers could nevertheless include them in an OO design. For the COTS components, the black box approach wasn't always successful. In some cases, the projects also had to consider the components' bugs, so they couldn't treat the components as black boxes.

The software architecture often mirrored the hardware architecture, which made the impact of changes in hardware easier to determine. Most cases involving complex systems employed a layered architecture pattern. These layers made it easier to deal with embedded systems' growing complexity.

Architecture specification. UML was the most commonly used notation for architectural modeling. On the higher abstraction levels, the specific meaning of UML notations in the architecture documentation should be clear to all stakeholders, which was not always the case. Some projects documented this in a reference architecture or architecture manual (we discuss these documents in more detail later).

The Rational Rose RealTime modeling tool (www.rational.com/products/rosert) lets developers create executable models and completely generate source code. A few projects tried this approach. One project completely generated reusable embedded software components from Rational Rose RealTime models. However, most of these projects used Rational Rose RealTime experimentally.

For creating UML diagrams, respondents frequently mentioned only two tools: Microsoft Visio and Rational Rose. Projects used these tools mostly for drawing rather than modeling. This means, for instance, that models weren't always syntactically correct and consistent.

Other well-known notations that projects used for architectural modeling were dataflow diagrams, entity-relationship diagrams, flowcharts, and Hatley-Pirbhai diagrams.[4] Projects often used diagrams based on these notations to clarify textual architectural descriptions in architecture documents. Some projects used more free-form box-line drawings to document and communicate designs and architectures.

One project used the Koala component model[5] to describe the software architecture. Compared to box-line drawings, the Koala component model's graphical notation has a more defined syntax. Koala provides interface

and component definition languages based on C syntax. A Koala architecture diagram specifies the interfaces that a component provides and requires. This project used Microsoft Visio to draw the Koala diagrams.

Projects often used pseudocode and pre- and postconditions to specify interfaces. Although this technique is more structured than natural language, the resulting specifications were mostly incomplete, with many implicit assumptions. This not only sometimes led to misunderstandings but also hampered the use of other techniques such as formal verification.

Some projects referred to a reference architecture or an architecture user manual. These documents defined the specific notations in architectural documents and explained which architectural concepts to use and how to specify them.

Architecture analysis. Most projects did not explicitly address architecture verification during design; those that did primarily used qualitative techniques. Few projects used quantitative techniques such as Petri nets or rate monotonic scheduling analysis.[6] One reason is that quantitative-analysis tools need detailed information. In practice, projects often used an architecture only as a vehicle for communication among stakeholders.

The most commonly employed qualitative techniques were reviews, meetings, and checklists. Another qualitative technique employed was scenario-based analysis. With this technique, a project can consider whether the proposed architecture supports different scenarios. By using different types of scenarios (for example, use scenarios and change scenarios), a project not only can validate that the architecture supports a certain functionality but also can verify qualities such as changeability.

The respondents typically felt that formal verification techniques were inapplicable in an industrial setting. They considered these techniques to be useful only in limited application areas such as communication protocols or parts of security-critical systems. The few projects that used Rational Rose RealTime were able to use simulation to verify and validate architectures.

Reuse. Reuse is often considered one of the most important advantages of development using architectural principles. By defining clean, clear interfaces and adopting a component-based development style, projects should be able to assemble new applications from reusable components.

In general, reuse was rather ad hoc. Projects reused requirements, design documents, and code from earlier, similar projects by copying them. This was because most products were based on previous products.

For highly specialized products, respondents felt that using configurable components from a component repository was impossible. Another issue that sometimes prevented reuse was the difficulty of estimating both a reuse approach's benefits and the effort to introduce it.

In some cases a project or company explicitly organized reuse. One company did this in combination with the Koala component model. The company applied this model together with a proprietary method for developing product families.

Some companies had adopted a product line approach to create a product line or family architecture. When adopting this approach, the companies often had to extract the product line architecture from existing product architectures and implementations. This is called *reverse architecting*.

In most cases, hardware platforms served as the basis for defining product lines, but sometimes market segments determined product lines. When a company truly followed a product line approach, architecture design took variability into account.

One company used a propriety software development method that enabled large-scale, multisite, and incremental software development. This method defined separate long-term architecture projects and subsystem projects. The company used the subsystems in short-term projects to instantiate products.

Another company had a special project that made reusable components for a certain subsystem of the product architecture. The company used Rational Rose RealTime to develop these components as executable models.

Some companies practiced reuse by developing general platforms on top of which they developed different products. This strategy is closely related to product lines, which are often defined per platform.

Discussion

You might well ask, are these results repre-

> **The respondents typically felt that formal verification techniques were inapplicable in an industrial setting.**

> **Making software development technologies more flexible can help make tailoring more attractive.**

sentative of the whole embedded software domain? By interviewing several respondents with different roles in each company, we tried to get a representative understanding of that company's embedded-software-development processes. The amount of new information gathered during successive interviews decreased. So, we concluded we did have a representative understanding for that company.

With respect to embedded software development in general, we believe that the large number of respondents and the companies' diversity of size, products, and country of origin make this inventory's results representative, for Europe at least. However, whether we can extend these results to other areas (for example, the US) is questionable.

Another point for discussion is that the methods, tools, and techniques the companies used were rather general software engineering technologies. We expected that the companies would use more specialized tools in this domain. Memory, power, and real-time requirements were far less prominent during software development than we expected. That's because most general software engineering technologies didn't have special features for dealing with these requirements. Tailoring can be a solution to this problem, but it involves much effort, and the result is often too specific to apply to other processes. Making software development technologies more flexible can help make tailoring more attractive. So, flexible software development technologies are necessary.

We noticed a relatively large gap between the inventory's results and the available software development technologies. Why isn't industry using many of these technologies? During the interviews, respondents mentioned several reasons. We look at three of them here.

The first reason is compliance with legacy. As we mentioned before, most projects didn't start from scratch. Developers always have to deal with this legacy, which means that the technology used in current projects should at least be compatible with the technology used in previous products. Also, companies can often use previous products' components in new products with few or no adaptations. This contradicts the top-down approach in Figure 1. Unlike with that approach, components at a detailed level are available from the start, before the new product's architecture is even de-

fined. This would suggest a bottom-up approach. However, because most available software development approaches are top-down, they don't address this issue.

Another reason is maturity. Most development methods are defined at a conceptual level; how to deploy and use them is unclear. When methods are past this conceptual stage and even have tool implementations, their maturity can prevent industry from using them. This was the case for some requirements management tools. Some respondents said that these tools weren't suited for managing the complex dependencies between requirements and other development artifacts, such as design and test documentation. Also, integrating these tools with existing solutions for other problems such as configuration management and change management was not straightforward.

The third reason is complexity. Complex development technologies require highly skilled software engineers to apply them. But the development process also involves stakeholders who aren't software practitioners. For instance, as we mentioned before, project team members might use architecture specifications to communicate with (external) stakeholders. These stakeholders often do not understand complex technology such as formal architecture description languages. Still, formal specifications are sometimes necessary—for example, in safety-critical systems. To make such highly complex technologies more applicable in industry, these technologies should integrate with more accepted and easy-to-understand technologies. Such a strategy will hide complexity.

We've discussed only some of the opportunities for improving embedded-software-engineering technologies for requirements engineering and architecture design. Our inventory also considered software engineering processes that we haven't presented in this article. We'll use all these results in MOOSE to define cases where new technologies can be applied and tailored to industrial needs. ⓜ

About the Authors

Bas Graaf is a PhD student in the Delft University of Technology's Software Engineering Unit. His research interest is software architecture design and verification, focusing on nonfunctional properties of software systems. He received his MS in computer science from the Delft University of Technology. Contact him at Mekelweg 4, 2628 CD Delft, Netherlands; b.s.graaf@ewi.tudelft.nl.

Marco Lormans is a PhD student in the Delft University of Technology's Software Engineering Unit. His research interest is the specification and management of requirements, with a special interest in nonfunctional properties of embedded systems. He received his MSc in computer science from the Delft University of Technology. Contact him at Mekelweg 4, 2628 CD Delft, Netherlands; m.lormans@ewi.tudelft.nl.

Hans Toetenel is an associate professor of software engineering at the Delft University of Technology. His research interest is the modeling and analysis of nonfunctional properties of embedded real-time systems. He received his PhD in computer science from the Delft University of Technology. He is a member of the International Federation for Information Processing Technical Committee 2 Working Group 2.4 (Software Implementation Technology) and Euromicro. Contact him at Mekelweg 4, 2628 CD Delft, Netherlands; w.j.toetenel@ewi.tudelft.nl.

References

1. L.D.J. Eggermont, ed., *Embedded Systems Roadmap 2002*, Technology Foundation, 2002; www.stw.nl/progress/ESroadmap/ESRversion1.pdf.
2. Y. Wang et al., "What the Software Industry Says about the Practices Modelled in Current Software Process Models?" *Proc. 25th Euromicro Conf.* (Euromicro 99), vol. 2, IEEE CS Press, 1999, pp. 162–168.
3. A.H. Eden and R. Kazman, "Architecture, Design, Implementation," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 2003), IEEE CS Press, 2003, pp. 149–159.
4. D.J. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1987.
5. R. van Ommering et al., "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, Mar. 2000, pp. 78–85.
6. C.L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, Jan. 1973, pp. 46–61.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

SOFTWARE ENGINEERING GLOSSARY

REVIEWS

California State University, Sacramento ■ thayer@csus.edu

■ Richard H. Thayer

reviewer: One who is not part of the development unit.

budget review: A formal meeting at which the monetary expenditures for a system or software engineering project are presented to the user, customer, or other interested parties for comment and approval. Expenditures are compared to the budget. Differences between budget estimates and actual expenditures are explained.

management review: A review to determine the project status as it applies to project plans, schedules, budget, staffing, training, and so on. Monetary expenditures are compared with the budget, and differences between the budget estimates and actual expenditures are explained. Completion dates are compared with the master schedule, and differences between the scheduled and actual completion dates are explained.

milestone review: 1. A project management review that is conducted at the completion of each of the software development lifecycle phases (a milestone), including requirements, preliminary-design, detailed-design, implementation, test, and sometimes installation and checkout phases. 2. A formal review of the management and technical progress of a development project.

software specification review: A milestone review conducted to finalize software configuration item requirements so that the developer can initiate preliminary software design. The SSR is conducted when software configuration item requirements have been sufficiently defined to evaluate the developer's responsiveness to an interpretation of the system-level or segment-level technical requirements. A successful SSR is predicated on the developer's determination that the software requirements specification and interface specifications form a satisfactory basis for proceeding into the preliminary design phase. [Military Std. 1521B-1985]

system design review: A system milestone review conducted when system requirements and the design approach are defined, in sufficient detail to ensure a technical understanding between developer and customer. Alternative design approaches and corresponding test requirements have been considered and the developer has defined and selected the required equipment, logistic support, personnel, procedural data, and facilities. The system segments are identified in the design specification; the hardware and software configuration items are identified in the requirements specifications.