TRENDS IN SYSTEMS AND SOFTWARE VARIABILITY

Jan Bosch, Chalmers University of Technology

Rafael Capilla, Rey Juan Carlos University

Rich Hilliard, consulting software systems architect

THE FOLLOWING STORY has happened thousands of times. A company puts a product on the market, and the product proves to be very successful. Customers use the product and it's almost perfect, but it needs some changes to make it really fit the context in which it's used. The company considers this and provides a customer-specific version. At the same time, on the basis of customer feedback, the company realizes that several customer segments would be better served with a product focused for each segment. So, the company ends up with a multitude of significantly similar product versions.

At this point, the company realizes that many of the required changes must be implemented for most or even all product versions and that implementing the same change multiple times is really inefficient, time-consuming, and error prone. This often results in the creation of a platform from which the different products and customer-specific versions can be derived. This significantly improves development efficiency. However, a new challenge enters the arena: managing the points in the platform where the product versions' functionalities differ—that is, *variability management*.

Variability management involves two key challenges. First, industrial reality shows that for successful platforms, the number of variation points, variants (alternatives that can be selected for a variation point), and dependencies between variation points and variants easily reaches staggering levels. We've seen cases with tens of thousands of variation points. The sheer number of variation points often results in having to allocate a rapidly growing percentage of the R&D budget to resolve

 0740-7459/15/$31.00 © 2015 IEEE

the complexities resulting from managing such variation. In addition, it often results in a situation in which no one in the company has a comprehensive overview of the available variability and consequently the maintenance in terms of removing obsolete variation points, changing binding times, and other tasks. If the company doesn't address this situation as part of technical-debt management, the cost of deriving new products from the platform could rival the cost of building each product from scratch.

The second challenge pertains particularly to embedded systems, which consist of mechanical, hardware, and software parts. The mechanical and hardware parts also exhibit variation. Such variation, however, differs considerably from software variability in that it tends to primarily involve the system's manufacturing stage and is concerned more with physical dimensioning and assembly than with system functionality. However, the need exists to define dependencies between the mechanical and hardware variations and the software variations. Several companies we've collaborated with are struggling with this, especially as the amount of software and the functionality it provides grow aggressively and as these systems' value is increasingly defined through software.

Software variability concerns all lifecycle phases from requirements elicitation to postdeployment and run time. In principle, in each phase,

- a variation point can be introduced,
- variants can be added,
- dependencies can be introduced,
- a specific variant can be bound to the variation point, and
- an already bound variant can be replaced with another variant.

In addition, the selection of a variant can affect the rest of the system owing to dependencies between the variation points and variants.

Software Variability Today

More than 20 years ago, Kyo-Chul Kang and his colleagues introduced the FODA (Feature-Oriented Domain Analysis) model.[1] FODA describes systems' visible properties in terms of product features, and it models such variability using variants (features that could be mandatory, optional, or alternative) and variation points representing logical relationships between variants. Moreover, FODA describes basic relationships between features. It uses **requires** and **excludes** constraints to delimit the variability's scope in space (the number of allowed products you can build) and to define the incompatibilities of infeasible products, often motivated by business and technical reasons.

Since the advent of software product lines (SPLs) for building multiple and related products in a given domain, variability models have increased in popularity, too. From 1998 to 2008, researchers proposed and implemented numerous extensions and enhancements to the original FODA model. Table 1 lists the most important contributions to software variability and includes proposals regarding the extension of feature models (FMs).[2]

Most of the FODA extensions in Table 1 focus on how to better represent a product family's *variability in space*. Most of them emphasize notational improvements, new types of features, cardinalities and feature attributes, and extended relationships to define more accurately the constraints and relationships between features.

However, other research has emphasized *variability in time* (also called *binding time*). Variability in time is a property of variability models and products that defines when features should or can be bound to their values to realize the products' allowed variability. Software designers can employ this property to delay design decisions. Although this research area encompasses fewer works than in Table 1, it becomes especially relevant to the dynamic features of modern software that exploit run-time configuration properties.

The evolution of complex systems tends to focus on dynamic aspects and on postdeployment configuration and reconfiguration. So, the binding time of SPL approaches has also evolved from static binding (for example, during design compilation, linking, or assembly) to dynamic binding, in which the variability is fully operationalized after deployment (for example, at run time).

Other intermediate binding modes, such as during configuration and installation, can be perceived as a combination of static and dynamic, depending on the variability realization mechanism. For instance,

TABLE 1

The evolution of variability models as Feature-Oriented Domain Analysis (FODA) extensions.

Variability approach	Authors	Year	Proposed extensions
FORM (Feature-Oriented Reuse Method)	Kang et al.[3]	1998	Feature viewpoints
FeatuRSEB (Featured Reuse-Driven Software Engineering Business)	Griss et al.[4]	1998	Notational changes Variation point features and variant features
Generative Programming FM (feature models)	Czarnecki and Eisenecker[5]	2000	Redefine an alternative relationship to OR/XOR
—	Hein et al.[6]	2000	UML-based Secondary structure for *require* dependencies
—	Van Gurp et al.[7]	2001	External features Redefine generalization and specialization relationships
—	Capilla and Dueñas[8]	2001	Cardinality and quantitative range of values Semantic relationships between features Quality-of-service features labeled
—	Riebisch et al.[9]	2002	Feature group Group cardinality
GP-extended (GP stands for generative programming)	Czarnecki et al.[10]	2002	Feature cardinality
Cardinality-based FM	Czarnecki et al.[11]	2004	Feature group Group cardinality
PLUSS (*P*roduct *L*ine *U*se Case Modeling for *S*ystems and *S*oftware Engineering)	Eriksson et al.[12]	2005	Notational changes
—	Benavides et al.[13]	2005	Feature attributes
OVM (Orthogonal Variability Modeling)	Pohl et al.[14]	2005	Graphical notation for variability of a software product line Internal and external variation points Traceability between variability and software artifacts
CVL (Common Variability Language)	Haugen et al.[15]	2008	Different kinds of variation points Language for expressing constraints

a software engineer can manually and statically configure product options before deployment. However, if the configuration uses either an automatic remote-update mechanism once the product is initially deployed or a dynamic library that automatically uploads a new configuration's values, the binding time can be considered dynamic.

Table 2 lists the most important binding-time approaches. It highlights the following four aspects concerning the binding time and the variability realization technique used (which is outside this article's scope).

First, the initial binding-time classification provided a way to classify binding-time modes according to the variability realization mechanism used.[16] Claudia Fritsch and her colleagues suggested run-time variability but didn't implement it because they focused largely on predeployment.

The second aspect is *feature-binding units* (FBUs), in which groups of related features bind to their values at the same time.[18] This clearly simplifies implementing the binding-time property and eases implementing and managing variability, particularly for the variation

points. This approach also facilitates the understanding of when features are activated or tracked for their dependencies and of the consistency of features in the same binding unit.

The third aspect is the realization of the variability at run-time binding modes as a postdeployment binding time.[19] Although this approach introduces implementation complexity, it eases management and evolution of the variability for unforeseen scenarios.

The final aspect suggests a detailed taxonomy for all binding-time modes and how to make transitions between binding times, which are needed for critical systems.[20]

TABLE 2

Binding-time approaches.

Binding-time approach	Authors	Year	Proposed FODA extension
—	Fritsch et al.[16]	2002	Binding-time classification
Variability at any time	Goedicke et al.[17]	2004	Run-time variation points
FBU (feature-binding units)	Lee and Muthig[18]	2008	Feature-binding units
Variants on the fly	Helleboogh et al.[19]	2009	Variability at run-time binding modes
—	Bosch and Capilla[20]	2012	Run-time-binding-mode taxonomy Multiple binding times Transitions between binding modes

Technical Practice Areas for Software Variability

A significant amount of variability research and practice deals with the representational aspects of variability in space and time. So, the following five practice areas are suitable for variability management.[21]

The first is *requirements variability*. Requirements are the entry level for expressing variability concerns. Requirements can vary from business-related requirements, to quality requirements, to technical requirements describing product properties. Little research has investigated software requirements variability—most solutions for modeling and managing variability concentrate on architecture and components. So, requirements variability is expressed only in terms of common requirements for the entire product line or requirements describing the variability within a specific product and the product's options for different customers' needs. Many software companies prefer to express product capabilities in terms of "features" rather than "requirements," which is closer to how designers understand and represent variability.

The second practice area is *architecture variability*. Because variability is reflected primarily in the architecture, this is our first practice area in which software variability must be represented alongside architectural artifacts.[22] However, current architecture modeling notations' lack of expressiveness makes it difficult to integrate current variability notation into software architecture descriptions. Also, this duality of different representation techniques and the lack of supporting tools has been a constant for many years and is a nightmare for software architects. So, variability management becomes a decision-oriented problem that can hardly be represented in the software architecture in a standardized way because variation points and variants lack explicit representation mechanisms in current architecture-modeling notations. Another factor complicating variability management in architecture is the cascading effect of decisions when variability models affect different architectural layers and, thereby, how the variants selected in upper layers are resolved at the implementation level.

The third practice area is *component development variability*. At the implementation level, variability must be implemented and realized in both reusable components and products. The decisions for selecting variants from design to implementation time must have a direct correspondence at the implementation level. However, the postdeployment selection of variants complicates the realization of variability. In that process, run-time concerns enable the selection of run-time binding modes because products can be reconfigured dynamically several times. The run-time realization of variability demands new mechanisms that allow the selection of variants and their options dynamically at any time.

The fourth practice area is *run-time variability*. Both researchers and practitioners tend to treat postdeployment variability, particularly run-time variability, differently from predeployment variability.[20] Research has shown that variation points tend to move to later and later binding times.[20] So,

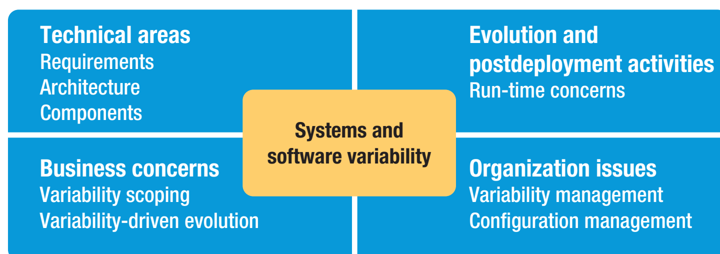


FIGURE 1. Aspects of variability management in industry. The diagram doesn't include the testing of variability models, which affects both product development and the checking of changes to constraints during product evolution.

many variation points that currently are bound before deployment will eventually become run-time variation points. Designing variation points such that the variability mechanism, which determines the binding time, can be easily replaced during system implementation is particularly important.

The final practice area, as the previous paragraph illustrates, is the *evolution of variability*. Variation points tend to be extended with variants later in the lifecycle, the binding tends to occur later, permanent binding tends to be replaced with more flexible alternatives, and so on. This requires replacing the variability mechanism, which is challenging in mature systems. In addition, maintaining variability is a challenge in that the number of variation points tends to increase because obsolete variation points often aren't removed. This leads to an increasingly complex system or platform, which affects the R&D organization's productivity and can decrease quality.

These are the most relevant technical areas in which variability must be represented in the different software artifacts. However, the management side of the problem, in which hundreds of variants must be captured, visualized, and modified, still becomes challenging for companies. This is because many of them lack full tool support for the development chain and use their own tools. Krzysztof Czarnecki and his colleagues summarized variability-modeling approaches understood as decision models and highlighted variability's dimensions.[23] Today, research on variability toolsets, languages, and frameworks in academia and industry is attempting to solve the variability management problem, including through automatic constraint solving. Examples of such research include FAMA (*Fea*ture *M*odel *A*nalyzer), FAMILIAR (*Fea*ture *M*odel Scr*i*pt *L*anguage for Ma*ni*pulation and *A*utomatic *R*easoning), Clafer (*cla*ss, *fe*ature, *r*eference), pure::variants, PLUM (Product Line Unified Modeller), and Gears.

An Industrial Perspective

Software variability is well recognized and supported in companies that have adopted an SPL approach to deliver their product portfolio faster and to maximize reuse. Industry has used very large feature models in successful product lines.[24] A recent study surveyed variability-modeling practice in companies in different countries.[25] Besides the organizational aspects demanded by an SPL approach, product configuration was the primary concern for variability modeling, whereas variability scoping was perceived as relevant for marketing purposes.

The survey also reported on variability notations for different domain-specific needs. Similarly, the companies were using a variety of tools with varying market penetration. Companies employing an SPL approach preferred commercial tools such as pure::variants, GEARS, and PLUM because those tools resulted from research that had been carried out for years. The heterogeneity of notations and tools shows that industry hasn't yet solved the variability management problem and continues to experiment with solutions and approaches. The lack of full integration of variability modeling into the development chain hampers broader, efficient use of variability in existing software production methods.

Figure 1 summarizes the issues most relevant to software variability in industry. It doesn't include the testing of variability models, which affects both product development and the checking of changes to constraints during product evolution.

Current and Future Trends

Companies from such varied domains as consumer electronics, the automotive sector, energy, telecommunications, and aviation have been investing significant effort incorporating software variability into their product line approaches. Increasing customer demand for configurable products, often after deployment, and the need for self-adaptive variants that realize run-time selection of a system's options have led

to requirements that conventional product lines and current variability mechanisms can't address. Companies that extensively use variability mechanisms and tools to manage hundreds of variants and support the evolution of static and dynamic product reconfigurations still lack powerful-enough solutions for managing the variability dynamically.

Other problems for variability management pertain to visualizing large variability models. Current commercial tools offer limited support for advanced visualization mechanisms to filter out the variability of individual engineering units. This leaves product line engineers with full variability models that are difficult to understand and manage.

Modeling and managing variability are also complicated by a plethora of product constraints that create a disruptive view of the feature model. In such views, crosscutting constraints add confusion to the visualization of feature variants and variation points. To date, the hundreds of constraints used to delimit the scope of allowed products are managed by SAT (satisfiability) solvers that can automatically resolve the variability model's consistency and validity. However, the resolution of these constraints often occurs offline, whereas many self-adaptive and run-time reconfigurable systems demand the means to resolve and change the constraints dynamically.

Consequently, the emerging paradigms of dynamic SPLs[26] and dynamic-variability mechanisms[27] constitute attempts to manage variability after system deployment and, in some cases, at execution time. Furthermore, some systems demand supporting different operational modes in which variants could be selected at run time to modify system behavior and provide a smooth transition between binding times. This is the case in which, for instance, a system might go from a normal operational mode to a maintenance mode, as the system's options are reconfigured and then redeployed dynamically.

> We need mechanisms that can model feature constraints and ease their "machine processability."

Finally, variability modeling has adapted to new challenges in which static feature models must incorporate contextual information. In some cases, feature models employ context features (using context information) to model system variability. In other cases, they describe how context features interact (that is, collaborative features). So, context features must be identified through context variability analysis and captured in feature models.[28] These and other issues must be tackled and supported by software variability.

Table 3 summarizes the trends in software variability and typical application areas or systems in which it can be used.

Future research in systems and software variability should address the following four topics.

The first is variability representation techniques. A neutral technique would ease the representation of variants and variation points in architecture. In addition, there are different approaches to and models for representing variability (for example, the Common Variability Language and Orthogonal Variability Modeling). Progress could accelerate if the community formed a consensus on describing variability models.

The second topic is the visualization and management of large variability models. Current tools lack adequate mechanisms to visualize variability models for different stakeholders, thus making variability harder to manage. Moreover, SPL practitioners should provide ways to manage large variability models, from explicit representation of variants and variation points to configuration and realization issues, particularly for reconfiguring variants at any time.

The third topic is constraint management. Constraints often create a nightmare for software designers because they crosscut feature models, and the resolution of valid feature models becomes computationally complex. We need mechanisms that can model feature constraints and ease their "machine processability," particularly for constraints that could be added and checked dynamically.

The final topic is postdeployment reconfiguration. As systems become increasingly dynamic, adaptive, and reconfigurable at any time, adaptation managers, combined with dynamic-variability mechanisms, will provide enhanced support to manage variants and constraints. They'll also provide improved postdeployment reconfiguration.

In This Issue

When preparing the special issue, we were committed to selecting

TABLE 3

Software variability trends and techniques for different application areas and systems.

Software variability trend	Techniques and tools used or under research	Application areas and systems
Run-time concerns	Dynamic variability Multiple binding times	Self-adaptive and autonomous systems (for example, robots, smart cities, and smart cars) Workflow and process reconfiguration Real-time critical systems (for example, power plants)
Visualization	Fish-eye views Filters Zooming Focus and context Cross-tree constraint views	Variability management tools
Run-time constraint checking	Online SAT (satisfiability) solvers Adaptation managers Complex constraints	Variability management tools Run-time self-adaptive systems (for example, autonomous robots)
Context and collaborative features	Context variability	Autonomous systems Drones and swarm systems Complex and critical systems-of-systems (for example, airport management systems)

high-quality articles that address different topics and trends of software variability management. Here we provide an overview of the selected articles.

In "Run-Time Variability for Context-Aware Smart Workflows," Aitor Murguzur and his colleagues describe LateVa (Late Variability for Context-Aware Smart Workflows), a framework for modeling and managing run-time variability in workflow systems. Their prototype provided run-time configuration of features to adapt automated-warehouse workflows to the current context (the workflow rate, types of sensors engaged, and physical properties of the boxes being moved).

In "A Reference Architecture and Knowledge-Based Structures for Smart Manufacturing Networks," Michael Papazoglou and his colleagues propose a Manufacturing Reference Architecture (MRA). The MRA supports demand-driven, collaborative manufacturing; a flexible production chain; and product customization. It is targeted at the emerging paradigm of smart manufacturing networks, adopting a marketplace-based approach. It comprises a logical organization of common software modules, and guidance for its extension to domain- or industry-shared platforms. The MRA also uses knowledge representation techniques to capture and share manufacturing information between phases and vendors, in the form of "blueprints." The authors illustrate their approach with an example from the automotive sector.

Techniques that originated in SPL development, including feature and variability modeling, are being applied to several of today's emerging cyber-physical-systems domains, including smart houses, logistics, smart manufacturing, and a plethora of smart and adaptive systems. The evolution of variability modeling and management techniques must incorporate new requirements to model the variability of modern systems with special run-time concerns. So, we've summarized here recent trends in software variability that provide opportunities for R&D activities. The proliferation of systems that demand postdeployment and reconfiguration tasks at any time brings new challenges for conventional variability management approaches and tools that SPL engineers must address.

References

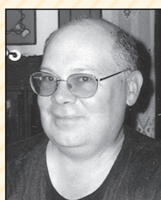
1. K.C. Kang et al., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, tech report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie Mellon Univ., 1990.
2. K.C. Kang and H. Lee, "Variability Modeling," *Systems and Software Variability Management: Concepts, Tools, and Experiences*, R. Capilla et al., eds., Springer, 2013, pp. 25–42.
3. K.C. Kang et al., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Eng.*, vol. 5, 1998, pp. 143–168.
4. M.L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," *Proc. 5th Int'l Conf. Software Reuse*, 1998, pp. 76–85.
5. K. Czarnecki and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

6. A. Hein, M. Schlick, and R. Vinga-Martins, "Applying Feature Models in Industrial Settings," *Proc. 1st Int'l Software Product Line Conf.*, 2000, pp. 47–70.
7. J. van Gurp, J. Bosch, and M. Svahnberg, "On the Notion of Variability in Software Product Lines," *Proc. 2001 Working IEEE/IFIP Conf. Software Architecture*, 2001, pp. 45–54.
8. R. Capilla and J.C. Dueñas, "Modelling Variability with Features in Distributed Architectures," *Software Product-Family Engineering*, LNCS 2290, Springer, 2001, pp. 319–329.
9. M. Riebisch et al., "Extending Feature Diagrams with UML Multiplicities," *Proc. 6th World Conf. Integrated Design & Process Technology*, 2002, pp. 1–7.
10. K. Czarnecki et al., "Generative Programming for Embedded Software: An Industrial Experience Report," *Generative Programming and Component Engineering*, LNCS 2487, Springer, 2002, pp. 156–172.
11. K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," *Software Product Lines*, LNCS 3154, Springer, 2004, pp. 266–283.
12. M. Eriksson, J. Börstler, and K. Borg, "The PLUSS Approach—Domain Modeling with Features, Use Cases and Use Case Realizations," *Software Product Lines*, LNCS 3714, Springer, 2005, pp. 33–44.
13. D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated Reasoning on Feature Models," *Advanced Information Systems Engineering*, LNCS 3520, Springer, 2005, pp. 491–503.
14. K. Pohl, G. Böckle, and F.J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, 2005.
15. O. Haugen et al., "Adding Standardized Variability to Domain Specific Languages," *Proc. 12th Int'l Software Product Line Conf.* (SPLC 08), 2008, pp. 139–148.
16. C. Fritsch et al., "Evaluating Variability Implementation Mechanisms," *Proc. Int'l Workshop Product Line Eng.* (PLEES), 2002, pp. 59–64.
17. M. Goedicke, C. Köllmann, and U. Zdun, "Designing Runtime Variation Points in Product Line Architectures: Three Cases," *Science of Computer Programming*, vol. 53, no. 3, 2004, pp. 353–380.
18. J. Lee and D. Muthig, "Feature-Oriented Analysis and Specification of Dynamic Product Reconfiguration," *High Confidence Software Reuse in Large Systems*, LNCS 5030, Springer, 2008, pp. 154–165.
19. D. Helleboogh et al., "Adding Variants on-the-Fly: Modeling Meta-variability in Dynamic Software Product Lines," *Proc. 3rd Int'l Workshop Dynamic Software Product Lines* (DSPL 09), 2009, pp. 18–27.
20. J. Bosch and R. Capilla, "Dynamic Variability in Software-Intensive Embedded System Families," *IEEE Software*, vol. 45, no. 10, 2012, pp. 28–35.
21. R. Capilla, J. Bosch, and K.C. Kang, *Systems and Software Variability Management: Concepts, Tools and Experiences*, Springer, 2013.
22. M. Galster et al., "Variability in Software Architecture: Current Practice and Challenges," *ACM SIGSOFT Software Eng. Notes*, vol. 36, no. 5, 2011, pp. 30–32.
23. K. Czarnecki et al., "Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches," *Proc. 6th Int'l Workshop Variability Modeling of Software-Intensive Systems* (VaMoS 12), 2012, pp. 173–182.
24. F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action*, Springer, 2007.
25. T. Berger et al., "A Survey of Variability Modeling in Industrial Practice," *Proc. 7th Int'l Workshop Variability Modeling of Software-Intensive Systems* (VaMoS 13), 2013, article 7.
26. M. Hinchey, S. Park, and K. Schmid, "Building Dynamic Software Product Lines," *Computer*, vol. 45, no. 10, 2012, pp. 22–26.
27. R. Capilla et al., "An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry," *J. Systems and Software*, May 2014, pp. 3–23.
28. H. Hartmann and T. Trew, "Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains," *Proc. 12th Int'l Software Product Line Conf.* (SPLC 08), 2008, pp. 12–21.

ABOUT THE AUTHORS

JAN BOSCH is a professor of software engineering at the Chalmers University of Technology. His research interests are software architecture assessment, design, and representation; software product lines; design erosion; component-oriented software engineering; and object-oriented frameworks and design patterns. Bosch received a PhD in computer science from Lund University. Contact him at jan@jan.bosch.com.

RAFAEL CAPILLA is an associate professor of computer science at Rey Juan Carlos University. His research interests are software architecture, software-product-line engineering, software variability management, and technical debt. Capilla received a PhD in computer science from Rey Juan Carlos University. Contact him at rafael.capilla@urjc.es.



RICH HILLIARD is a software systems architect, consulting with public- and private-sector clients. He's the project editor of ISO/IEC/IEEE 42010, Systems and software engineering—Architecture description (the international adoption of IEEE Std. 1471-2000). Hilliard is a visiting research scientist at MIT's Experimental Study Group. He's the vice-chair of IFIP Working Group 2.10 on Software Architecture and a member of the Free Software Foundation and IEEE Computer Society. Contact him at richh@mit.edu; http://softsysarchitect.net.

See **www.computer.org/software-multimedia** for multimedia content related to this article.