Your "What" Is My "How": Iteration and Hierarchy in System Design

Michael W. Whalen, University of Minnesota

Andrew Gacek and Darren Cofer, Rockwell Collins

Anitha Murugesan, Mats P.E. Heimdahl, and Sanjai Rayadurgam, University of Minnesota

// Requirements and architectural design should be more closely aligned than they currently are: requirements models must account for hierarchical system construction, and architectural design notations must better support requirements specification for system components. //

CONSIDER A MODERN aircraft, such as a Boeing 747: it's an extraordinarily complex system containing more than 6 million parts, 171 miles of wiring, and five miles of tubing (www.boeing.com/commercial/747family/pf/pf_facts.html). Viewed another way, it's a complex software and hardware infrastructure containing 6.5 MLOC distributed across dozens of different computing resources.[1] To make design and construction possible, the components—physical and software—are necessarily organized as a hierarchical federation of systems that interact to satisfy the aircraft's safety, reliability, and performance requirements.

This hierarchical aspect of design is crucial. Design considerations at one level of abstraction, such as in partitioning a system into subsystems and allocating functionality to each, determine what the subsystems should do at the next level of abstraction. Requirements at a particular level in the hierarchy are implemented in terms of a set of design decisions (an architecture), which in turn induces sets of requirements on that architecture's components; this is an idea that spans both physical and software architectures. Yet, we frequently speak of a system's "requirements" as separate from and more abstract than its "architecture."

Although some requirements engineering techniques do support hierarchical decomposition of requirements (notably, KAOS[2] and i*[3]), these decompositions generally aren't bound to the system's architecture, nor is there a prescribed process for coevolution with architectural models. Therefore, when practitioners derive an architecture to address a systems engineering challenge, they often have little guidance on how the requirements should be decomposed and allocated to architectural components.

Iterative Requirements and Architecture

Even in safety-critical systems with well-understood domains, it's difficult to correctly specify requirements. In previous work involving requirements verification in model-based development, we found that the requirements were almost as likely to be incorrect as the models.[4] For example, one class of errors involves inconsistencies between two requirements:

- When button X is pressed, the mode shall be A.
- When button Y is pressed, the mode shall be B.

These requirements are inconsistent if X and Y can be simultaneously pressed and A and B are mutually exclusive.

0740-7459/13/$31.00 © 2013 IEEE

By constructing and analyzing models, we were able to find such inconsistencies, as well as implicit assumptions about the environment in which the system was to be deployed. In fact, because the models regularly brought problems in the requirements to light, the engineers iteratively refined models and requirements using a "model a little, test a little" approach.

For very large systems (or systems of systems), it would be even less likely that the top-level requirements would be correct.[5] But assuming they actually are correct, an additional challenge is demonstrating that, given the architectural solution, the hierarchically decomposed requirements meet the system-level requirements. An approach to requirements validation, architectural design, and architectural verification that uses the requirements to drive the architectural decomposition and the architecture to iteratively validate the requirements would be highly desirable. Furthermore, we would like this verification and validation to occur prior to building code-level implementations.

A well-defined set of requirements comes via informed deliberations among stakeholders with shared as well as competing interests. We can view the starting point as an incomplete articulation of their key concerns. Over the course of these deliberations, the participants make rational choices and trade-offs. The resulting requirements' quality largely reflects how well the participants engaged in this process. Key stakeholders include the systems, safety, and software engineers, whose overriding concern is how to successfully build a system that would meet the resulting requirements. Understanding the architecture is essential for these stakeholders to determine the main concerns that should inform their positions during negotiations. It highlights important aspects of how the system would work, leaving out the minutiae and focusing participants' attention on concerns likely to affect system feasibility. Therefore, it's only natural that in any practical development process, requirements and architecture evolve together.

In this respect, we concur with Bashar Nuseibeh's twin peaks model,[6] which recognizes that requirements and architecture coevolve and that this helps create both a sound architecture and correct requirements. Nuseibeh also points out that system development often starts from candidate architectures that have been used in similar systems. Such architectures might restrict the set of achievable requirements but still be desirable for many reasons, including designers' and software engineers' familiarity with the architecture and amortization of cost owing to the candidate architecture getting refined over several systems. Thus, iteration between architectural models and requirements can better deal with key sources of requirements uncertainty identified by Barry Boehm[7]: the use of COTS components, "I'll Know It When I See It" (IKIWISI), and rapidly changing requirements.

In this article, we extend this view further and posit both that the dynamic model of coevolution induces a static model of interrelationship that ties requirements with architectural elements in an inherently hierarchical fashion, and that such a mapping is equally essential for both building and verifying complex systems.

Organizing Requirements

Once systems become sufficiently complex, they are decomposed into subsystems that are implemented by several distinct teams. Consequently, the requirements on the system as a whole must be decomposed and allocated to each of those subsystems. This decomposition affects both requirements and architecture because the decomposition's structure will influence how requirements flow down to each subsystem. Therefore, requirements should be organized into hierarchies that follow the system's architectural decomposition. This organization promotes a natural notion of refinement and traceability between layers of requirements.

Such organization highlights the idea that system decomposition is both an architectural and requirements exercise. The act of decomposing a system into components (and then assembling the components into a system) induces a requirements analysis effort in which we must ascertain whether the requirements allocated to subcomponents in the architecture are sufficient to establish the system-level requirements. Of equal importance, we must determine whether any assumptions on a component's environment made when allocating requirements to that component can be established (see Figure 1). As we begin to allocate requirements to components, we might find that the architecture we've chosen simply can't meet the system-level requirements. This might cause us to rearchitect the system to meet the system-level requirement, levy additional constraints on the

In any practical development process, requirements and architecture evolve together.

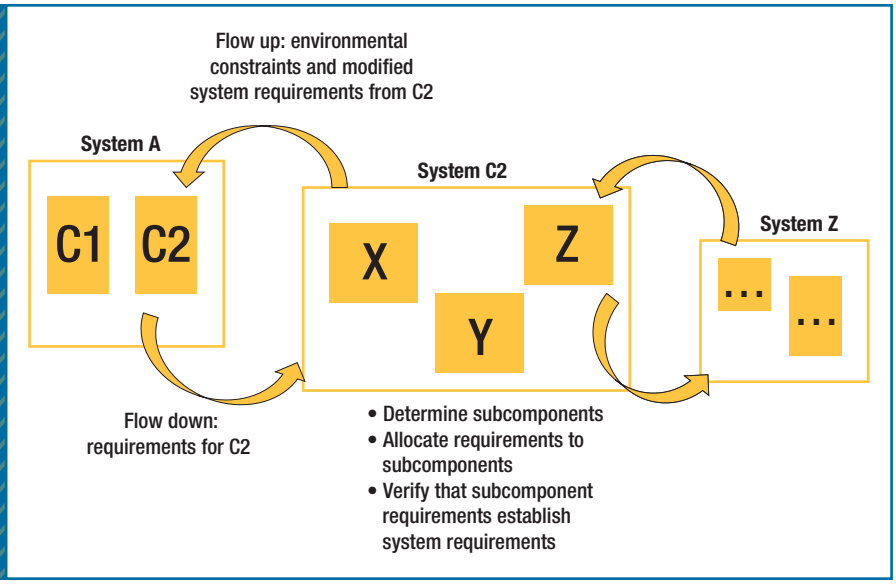

FIGURE 1. Interplay between architecture and requirements. This figure illustrates how requirements can flow both downward (for example, due to system decomposition) and upward (for example, due to use of COTS components).

external environment, or renegotiate the system-level requirement.

Architectural Models

Architectural models include components as well as those components' interconnections, interfaces, and requirements (but not their implementations). By annotating models with requirements for component behavior, they become a means to support iteration between requirements allocation and architectural design.

At the leaf level, component implementations are defined separately using model-based development tools or traditional programming languages, as appropriate. They're represented in the system model by the subset of specifications needed to describe their system-level interactions; these specifications can include information about component functionality, performance, security, bindings to hardware, and other concerns.

All embedded safety-critical systems require an architectural modeling language that can support descriptions of both hardware and software components and their interactions. We've considered both the Systems Modeling Language (SysML)[8] and Architecture Analysis and Design Language (AADL)[9] notations. SysML was designed for modeling the full scope of a system, including its users and the physical world, whereas AADL was designed for modeling real-time embedded systems. Although both SysML and AADL are extensible and can be tailored to support either domain, the fundamental constructs that each provides reflect these differences. For example, AADL lacks many of the constructs for eliciting system requirements such as SysML requirement diagrams and use cases. On the other hand, SysML lacks many of the constructs needed to model embedded systems such as processes, threads, processors, buses, and memory. Our approach has been to use AADL as our working notation and support translation from SysML (with some additional stereotypes for certain components corresponding to AADL constructs) into AADL models.

System Verification

In critical systems, there's been significant progress in analyzing the behavior of leaf-level components against their requirements. In the 2000s, tools and techniques for unit testing source code improved dramatically; today, coding errors that escape detection through testing are relatively rare.[10] During the past decade, model-based development has increased the level of abstraction at which engineers design software components and has moved much of the testing forward into the design phase. Also during this time period, model checking has become a practical form of analysis that finds errors that testing would miss and does so earlier in the design process.[11]

Although engineers have become better at demonstrating that leaf-level components meet their requirements, checking whether component-level requirements demonstrate the satisfaction of higher-level requirements is still an area of ongoing research. Not surprisingly, component integration has become the most significant source of errors in systems.[5] In fact, although techniques for specifying and verifying individual components have become highly automated, the most common tools used to specify the complex system architectures containing the components remain word processors, spreadsheets, and drawing packages. It will be important to develop better support for decomposition of requirements throughout the system architecture and subsequent verification that such decompositions are sound.

In the initial stages of requirements and architectural co-design, the process is relatively informal and fluid. However, for critical systems, such informality can lead to problems. Often, many of the errors in system

development manifest themselves in integration; each of the leaf-level components meets its requirements, but this isn't sufficient to establish system requirement satisfation. To prevent these integration errors, we wish to perform *virtual integration*, in which we can determine whether leaf-level requirements demonstrate the satisfaction of system level requirements at arbitrary levels of abstraction.

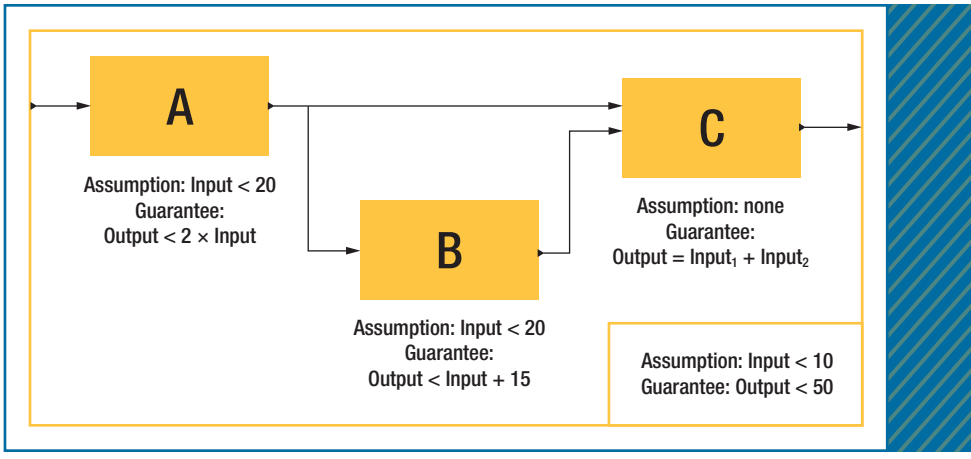

FIGURE 2. A tiny system architecture to illustrate assume-guarantee contracts.

Foundations

When we base the requirements and architecture efforts on natural-language requirements and modeling notations lacking rigorous semantics, the reasoning process we promote closely resembles the satisfaction argument of Jonathan Hammond and his colleagues.[12] The satisfaction argument looks to establish that system requirements hold through an argument involving the system behavior specification and assumptions about the system domain. When systems are decomposed, a subcomponent's domain assumptions will likely include assumptions about the behaviors of the other subcomponents with which it communicates.

To formalize satisfaction arguments, provide an appropriate mechanism for capturing needed information from other modeling domains to reason about system-level properties.[13] In this formulation, guarantees correspond to component requirements and are verified separately as part of the component development process, either by formal methods (such as model checking) or traditional means involving testing and inspections. Assumptions correspond to the environmental constraints that were used to verify that the component satisfies its requirements. For formally verified components, they are the assertions or invariants on the component inputs that were used in the proof process. A contract specifies precisely the information needed to reason about the component's interaction with other parts of the system. Furthermore, a contract mechanism supports a hierarchical decomposition of verification processes that follows the system model's natural hierarchy.

The idea is that, for a given layer of the architecture, we use the contracts of the subcomponents within the architecture to satisfy the system-level requirements allocated to that level. Figure 2 shows a simplified example of the idea. Here, we want to establish at the system level that the output signal is always less than 50 when the input signal is less than 10. We can prove this using the assumptions and guarantees provided by subcomponents A, B, and C. This figure shows one layer of decomposition, but the idea generalizes arbitrarily to many layers. To create a complete proof, we must prove that each layer establishes its system-level property.

The system-level properties that we wish to verify fall into several categories requiring different verification approaches and tools. At the topmost level, we're interested in behavioral properties that describe the state of the system as it changes over time. Behavioral properties describe protocols governing component interactions in the system or the system's response to combinations of triggering events. Currently, we use the Property Specification Language (PSL) to specify most behavioral properties of components. This allows straightforward formulation of a variety of temporal logic properties. Recently, Rockwell Collins and the University of Minnesota created a relevant tool suite called the assume guarantee reasoning environment (AGREE), which we describe in greater detail elsewhere.[14]

Goals

We had two goals in creating this verification approach: the first was to reuse the verification already performed on components, and the second was to enable distributed development by establishing the formal requirements of subcomponents that are used to assemble a system architecture. If we can establish a system property of interest using the contracts of its components, then we have a means of performing virtual integration of components. We can use the contract of each of the components as a specification for suppliers, giving us a great deal of confidence that if all the suppliers meet the specifications, the integrated system will work properly. Thus, we can arbitrarily choose

the leaf level of the components (and their requirements) that we wish to analyze.

Figure 2 illustrates the compositional verification conditions for a toy example. Components are organized hierarchically into systems. We want to be able to compose proofs starting from the leaf components (those whose implementation is specified outside of the architecture model) recursively through all the layers of the architecture. Each layer of the architecture is considered to be a system with inputs and outputs and containing a collection of components. A system S can be described by its own contract (A_S, P_S) plus the contracts of its components C_S, so S = (A_S, P_S, C_S). Components communicate in the sense that their formulas can refer to the same variables. For a given layer, the proof obligation is to demonstrate that the system guarantee P_S is provable given the behavior of its subcomponents C_S and the system assumption A_S—that is, we should be able to derive P_S as a consequence of C_S and A_S by applying the rules of the logic used to formulate these contracts. Such a proof, in effect, assures a successful integration of the contract-conforming components to realize a system that can meet its contract, reducing both the burden and risk associated with system integration during development.

In our framework, we use past-time linear temporal logic (PLTL) to formulate systems' correctness obligations. Temporal logics such as PLTL include operators for reasoning about the behavior of propositions over a sequence of instants in time. For example, to say that property P is always true at every instant in time (that is, that it's globally true), we would use G(P), where G stands for "globally." The correctness obligations are the form G(H(A) ⇒ P), which informally means that if assumption A has been true from the beginning of the execution up until this instant (that is, assumption A is historically true), then guarantee P is true.

For the obligation in Figure 2, our goal is to prove the formula $G(H(A_S) \Rightarrow P_S)$ given the contracted behavior $G(H(A_C) \Rightarrow P_C)$ for each component c within the system. It's conceivable that for a given system instance, a sufficiently powerful model checker could prove this goal directly from the system and component assumptions. However, we take a more general approach: we establish generic verification conditions that together are sufficient to establish the goal formula. In this example, this means that for system S, we want to prove that output < 50 assuming that input < 10, and that the contracts for components A, B, and C are satisfied. For a system with n components, there are $n + 1$ verification conditions: one for each component and one for the system as a whole. The component verification conditions establish that each component's assumptions are implied by the system-level assumptions and the properties of the sibling components. For this system, the verification conditions generated would be

$$G(H(A_S) \Rightarrow A_A)$$

$$G(H(A_S \wedge P_A) \Rightarrow A_B)$$

$$G(H(A_S \wedge P_A \wedge P_B) \Rightarrow A_C)$$

$$G(H(A_S \wedge P_A \wedge P_B \wedge P_C) \Rightarrow P_S).$$

In general, these architectures can contain cycles between components, in which component A requires the guarantees of component B and vice versa, which can lead to unsound circular reasoning. To avoid this, we use induction over time, which requires that (at least) one of the components can only refer to guarantees of the other in earlier instants in time. This ensures that at a given instant in time, there is no circularity.[13] The system-level verification condition shows that the system guarantees follow from the system assumptions and each subcomponent's properties. This is essentially an expansion of the original goal $G(H(A_S) \Rightarrow P_S)$, with the additional information obtained from each component.

Scaling to Real Systems

Of course, reasoning about toy examples is neither interesting nor useful for practitioners attempting to build large-scale systems. For a DoD-sponsored project, we modeled an avionics system architecture involving an autopilot, two redundant flight guidance systems, and a variety of redundant sensors. (Figure 3 shows the architecture's top layer.) Using this model, we proved properties describing limits on the transient commanded pitch behavior of the flight control system using AGREE.[14] Even given a relatively complex architecture, each compositional analysis required a small amount of time owing to the analysis problem's decomposition into layers—on the order of five seconds for each layer of the avionics system.

An important limitation in the current tool suite[14] is that it can only deal with systems that are synchronous with a one-step communication delay between connected components. The synchrony hypothesis, in this case, means that the components share a global clock. In order to be appropriate for full-scale use, we must accurately support notions of time in our composition framework. This isn't likely to require any changes to the underlying formalism of composition, but we must account for the delays induced by computation

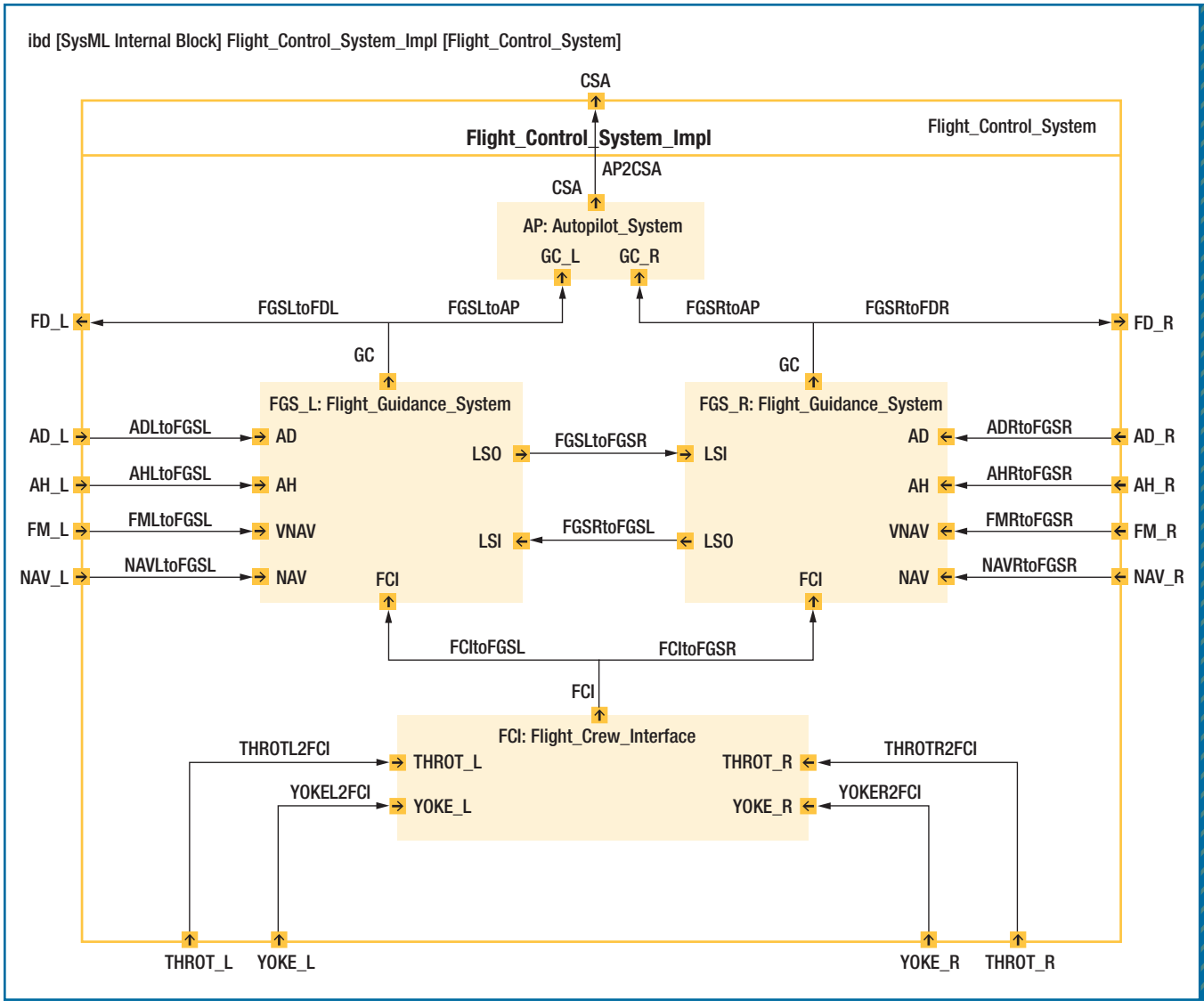

FIGURE 3. A fragment of an avionics flight control system modeled in SysML containing dual-redundant flight guidance systems.

time, network traffic, and other architectural properties of the model through extraction of this information from the AADL model and incorporation into the formal analysis model. PSL provides some support because it lets us add property clocks, representing the instants at which they should be examined. We can use these clocks to describe instants in which a component operates in the context of a larger system. This is the major focus of the next phase of our work, in which we will be modeling more realistic avionics and medical device architectures.

A more general concern regards the choice of representing components as sets of PSL properties as opposed to other formalisms, such as process algebras. In our work, we have found that declarative properties can be closely aligned with a style of requirements that are traditionally used in avionics systems.[4] However, complex coordination activities among multiple components within an architecture can be difficult to represent using temporal logic. In future work, we hope to examine whether the process-algebraic view of the system can be aligned with our temporal-logic view.

Acknowledgments

An earlier version of this article as a position paper is included in the *Proceedings of First International Workshop on the Twin Peaks*

ABOUT THE AUTHORS

MICHAEL M. WHALEN is a program director at the University of Minnesota Software Engineering Center. His research interests include formal analysis, language translation, testing, and requirements engineering. Whalen received a PhD in computer science from the University of Minnesota. He's a senior member of IEEE. Contact him at whalen@cs.umn.edu.

ANDREW GACEK is a senior systems engineer in Rockwell Collins' Trusted Systems group. His research interests include automated testing, model checking, and logic. Gacek received a PhD in programming languages and logic from the University of Minnesota. Contact him at ajgacek@rockwellcollins.com.

DARREN COFER is a principal systems engineer at Rockwell Collins' Advanced Technology Center. His research interests include the use of formal methods for verification and certification of high-assurance embedded systems. Cofer received a PhD in electrical and computer engineering from the University of Texas at Austin. He's a senior member of IEEE. Contact him at ddcofer@rockwellcollins.com.

ANITHA MURUGESAN is a PhD student in software engineering at the University of Minnesota. Her research interests include requirements engineering and modeling for cyberphysical systems. Murugesan received an MTech in computer science and engineering from Vellore Institute of Technology. Contact her at anitha@cs.umn.edu.

MATS P.E. HEIMDAHL is a full professor of computer science and engineering at the University of Minnesota. His research interests include software engineering, safety-critical systems, testing, requirements engineering, and automated analysis of specifications. Heimdahl received a PhD in information and computer science from the University of California, Irvine. He's a member of IEEE, ACM, and PSIA. Contact him at heimdahl@cs.umn.edu.

SANJAI RAYADURGAM is a program director at the University of Minnesota Software Engineering Center. His research interests include software testing, formal analysis, and requirements modeling, with particular focus on safety-critical systems development. Rayadurgam received a PhD from the University of Minnesota at Twin Cities. He's a member of IEEE and ACM. Contact him at rsanjai@cs.umn.edu.

of Requirements and Architecture, IEEE, 2012, pp. 36–40. DARPA/AFRL (on project FA8650-10-C-7081) and NSF grants CNS-0931931 and CNS-1035715 have partially supported this work.

References

1. R. Charette, "This Car Runs on Code," *IEEE Spectrum*, Feb. 2009; http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code.
2. A. van Lamsweerde, "Engineering Requirements for System Reliability and Security," *Software System Reliability and Security*, M. Broy, J. Grunbauer, and C.A.R. Hoare, eds., vol. 9, IOS Press, 2007, pp. 196–238.
3. E. Yu et al., *Social Modeling for Requirements Engineering*, MIT Press, 2011.
4. S.P. Miller et al., "Proving the Shalls: Early Validation of Requirements through Formal Methods," *Int'l J. Software. Tools for Technology Transfer*, vol. 8, no. 4, 2006, pp. 303–319.
5. R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc. IEEE Int'l Symp. Requirements Engineering*, IEEE, 1993, pp. 126–133.
6. B. Nuseibeh, "Weaving Together Requirements and Architectures," *Computer*, vol. 34, no. 3, 2001, pp. 115–117.
7. B. Boehm, "Requirements that Handle IKIWISI, COTS, and Rapid Change," *Computer*, vol. 33, no. 7, 2000, pp. 99–102.
8. S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: Systems Modeling Language*, Morgan Kaufmann, 2008.
9. *Std. SAE-AS5506, Architecture Analysis and Design Language*, SAE Int'l, Nov. 2004.
10. J. Rushby, "New Challenges in Certification for Aircraft Software," *Proc. 9th ACM Int'l Conf. Embedded Software*, ACM, 2011, pp. 211–218.
11. S.P. Miller, M.W. Whalen, and D.D. Cofer, "Software Model Checking Takes Off," *Comm. ACM*, vol. 53, no. 2, 2010, pp. 58–64.
12. J. Hammond, R. Rawlings, and A. Hall, "Will It Work?," *Proc. 5th IEEE Int'l Symp. Requirements Eng.*, IEEE, 2001, pp. 102–109.
13. K.L. McMillan, "Circular Compositional Reasoning about Liveness," tech. report 1999-02, Cadence Berkeley Labs, 1999.
14. D.D. Cofer et al., "Compositional Verification of Architectural Models," *Proc. 4th NASA Formal Methods Symp.* (NFM 12), A.E. Goodloe and S. Person, eds., vol. 7226. Springer, 2012, pp. 126–140.