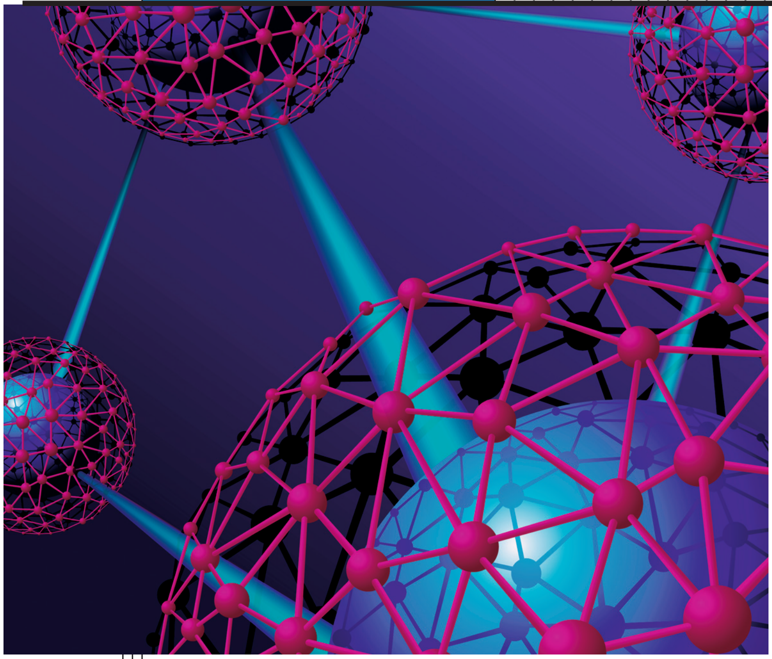

EVOLVING EMBEDDED SYSTEMS

Gabor Karsai, *Vanderbilt University*
Fabio Massacci, *University of Trento, Italy*
Leon J. Osterweil, *University of Massachusetts Amherst*
Ina Schieferdecker, *Fraunhofer FOKUS and Technical University Berlin, Germany*

Integrated and embedded systems have become an invisible yet crucial part of our daily lives, making their continuous and trouble-free evolution of great importance.

Embedded systems span a wide range of domains, from household applications in appliances, entertainment devices, and vehicles to critical applications in patient-monitoring systems, industrial automation, and command-and-control systems. Several specific drivers can shape an embedded system's evolution. Many consumer-oriented systems, for example, undergo rapid changes because of market pressures to come up with new products or improve capabilities in existing ones. Another driver is hardware obsolescence—for example, a particular hardware component might need replacement, or new special-purpose hardware might replace software functions. Existing platforms might also need additional functions: if an embedded system vendor identifies a novel business opportunity, it might have to update existing and deployed systems to capitalize on that opportunity. Finally, users often invent new ways to manipulate existing systems, either to meet changing needs or because increased expertise causes them to seek more operational options.

All these drivers can lead to specific technical problems for embedded systems:

- Finely crafted and optimized designs must be maintained and evolved with great care—mass-produced embedded systems, for example, are often optimized to minimize resource consumption, and evolutionary modifications must avoid violating this property.
- Safety- and mission-critical embedded systems require system verification in some form, and this can be a bottleneck—the current practice of complete reverification is very expensive.
- The evolution process itself must be optimized because it's typically performed under a tight deadline; moreover, any change must be minimal, yet maximally effective, to meet the previous two challenges.
- Because embedded systems are often deployed in critical applications, they must evolve in vivo—they can't go offline for a long time.

These challenges can best be met through a combination of techniques and technologies. We discuss evolution on different timescales and in the context of user processes, load-time verification, and tests for checking system correctness. Our goal is to give a broad overview of relevant embedded system issues and some potential solutions.

EVOLUTIONARY TIMESCALES

System evolution can occur on multiple timescales.

Design time

Design-time evolution (DTE) offers many potential subjects for evolution. First, system design can evolve because of changes in requirements, the need for improvements, or the need to fix deficiencies. Second, system implementation can evolve—sometimes in concert with design, sometimes independently. Third, the tools used to create and analyze the design and implementation can evolve, although at a price: they might force developers to modify their designs or implementations to comply with new versions of tools. In extreme cases, the design or even the implementation language can change, triggering the problem of carrying forward existing engineering artifacts.

Tool support can help address DTE, but although research tools are available, industrial-quality tools aren't quite there yet. One key problem is the need to preserve or evolve design abstractions that may or may not be explicit in a design and are very rarely explicit in the implementation. If designs are represented as models in, for instance, UML, then transformation-based approaches could be useful.[1] Model-based, generative approaches offer an opportunity to facilitate evolution because models can typically be manipulated programmatically through an API and are on a much higher level of abstraction than code. However, designers still need tool-supported, higher-order techniques such as model transformations to express their intent. Many modern development environments now offer assistance with code refactoring, but design refactoring support is often lacking.

There are serious challenges in evolving the design and implementation of embedded systems—careless modifications can lead to major rework. One problem stems from the embedded code's emergent, nonfunctional properties: memory footprint, execution time, and stack usage are all difficult to estimate directly from the design. Thus, when the design or implementation changes, developers must determine these emergent properties (possibly through simulation and testing), and if they're unsatisfactory, revise the changes, which can lead to extensive and expensive iterations.

Another problem comes from the need to verify the embedded code that actually runs on the execution platform. Verifying code is difficult for a regular system, but for an embedded one it's even more complex because the code doesn't run in isolation, but on an execution platform whose properties must be explicitly known. Evolving an embedded system also means evolving the "proofs" about its correctness.

Load time

Load-time evolution (LTE) occurs when a system evolves in the field but is not in active operation. It is sometimes viewed as an operator-induced change in a system's configuration, but the change could be quite complex and lead to a new, "evolved" system. For instance, it's now customary for mobile phone users to download new applications that can connect to a GPS satellite and send their current geographical coordinates over the Internet to a social networking site: a major "evolution" in the phone's software.

The key word is evolution: making the system better to satisfy some optimization function.

The main question of LTE in embedded systems is again verification: how to prove that the evolved system is correct. This is important because fixing embedded systems in the field could be quite expensive. Another relevant question is how the evolution happens if it is user-driven instead of vendor-driven. Users aren't interested in low-level changes—they want specific system features and capabilities. An "LTE agent," or built-in system tool that translates user preferences and system constraints into low-level evolutionary changes on the system, could be a solution here.

Runtime

Runtime evolution (RTE) means changing the system while it is in active use. The evolutionary process is triggered by a system-made observation, possibly involving reflection and reasoning on the system's behalf. Few such systems exist today, but autonomic computing and autonomous vehicles offer some examples. The key word is evolution: making the system better to satisfy some optimization function. RTE is a deliberated and reasoned choice for change made by the system itself toward a new mode that improves it. What the system evolves to isn't necessarily predefined; rather, it's computed on the fly according to the current system state and environment.

Naturally, engineering RTE in systems is challenging, and the problems are well-known: What is the RTE's expected and allowed scope? How does the system detect the need for evolution? How does the system reason about what to evolve to? How is the actual evolution executed? How does the system verify the evolutionary step? What's a human user's role in the process? These questions are especially acute for embedded systems because of their often critical, resource-constrained, and closed nature. Perhaps the biggest challenge of all is how to ensure the dependability of embedded systems that evolve at runtime. Some recent research roadmaps and early results come primarily from the area of self-adaptive systems.[2]

CONCURRENT EVOLUTION OF SYSTEMS AND PROCESSES

Any system in use today will experience pressure to evolve by the very fact of its being in use, which implies that it meets—at least to some extent—real-world needs. This is particularly true for embedded systems because their very definition implies that they participate in real-world activities and processes. Most successful processes tend to allocate rote and mechanical tasks to software components in embedded systems, leaving humans to do relatively more creative work that requires insight and intelligence. Thus, successful embedded systems typically tend to grow in scope and power by taking on increasingly large quantities of rote and mechanical work. But in so doing, it isn't unusual for new mechanical and software capabilities to facilitate new exercises of human intelligence and creativity. Thus begins a cycle: real-world processes levy strong requirements on the embedded systems that they use, and as the embedded software components in these systems meet these requirements, they create pressures on the processes themselves to absorb more tasks. We can expect this cycle to continue indefinitely, as long as the embedded system and its software components experience actual use.

Evolving embedded systems requires a careful combination of verification and testing methods for development, load, and runtime evolution. For efficient online verification and validation, trusted and untrusted software is to be treated separately.

A key challenge for embedded systems is to continually provide satisfactory services, even as they strive to provide even more satisfactory capabilities. To do this, embedded systems and the software components that they contain must always demonstrably respond to an understood and agreed-upon set of requirements. Typically, these requirements are derived principally from the processes in which they're used. Thus, for example, a surgical process can impose specific requirements on the behaviors of doctors and nurses, but also on devices such as infusion pumps that are used in the process. The requirements imposed on the infusion pump itself are passed down to the software embedded in the pump as well.

Ultimately, embedded systems and their software components can't be considered to be absolutely correct or satisfactory. Such systems can only be judged to be correct or satisfactory relative to how well they meet the requirements imposed on them by the processes using them. An embedded system's participation in a process can also change expectations and desires. For example, using a powerful vote-recording device in an election process might cause poll workers to decide that they would indeed like the device to check for duplicate voters, even though the current process mandates that they perform this task themselves. However, such desires shouldn't be translated into actual process changes unless all participants' behaviors have changed to conform to the new process requirements. Thus, poll workers shouldn't stop performing manual checks to meet stronger security requirements—at least not until software embedded in the vote-recording device can address this requirement.

The need to synchronize process participant behavior with process requirements must focus attention on how to determine consistency. Technical approaches such as model checking[3,4] have proven to be effective in demonstrating the consistency (or lack thereof) of bodies of code or design with certain kinds of required properties. What's missing is a way to take process requirements and derive from them requirements for process participant behavior. Rigorous process definitions can best address this need. Experience with the Little-JIL language[5] suggests that this is quite feasible, although a wide range of other languages could also serve as effective bases for rigorously defining processes. The next step is for technologies to help take such definitions and derive requirements on process participant behavior from them. These requirements can then be used as the basis for verifying and testing embedded software. Approaches such as assume-guarantee-reasoning[6] and model-carrying code[7] (or its modern variants[8]) offer some promise of effectively supporting this capability.

VERIFICATION FOR LOAD-TIME EVOLUTION

A successful process that uses embedded systems can drive an evolutionary change, but the processes themselves shouldn't change until it's safe for them to do so. For example, the success of applications running on smart cards has led directly to a desire for smarter cards on which more than one application can run. Owners of different trust domains—banking, transportation, healthcare, telecommunications, and so on—want just one card on which they can load and update their applications asynchronously and independently from one another. Yet this change in process requirements also changes the requirements for the installation process. In addition to independent updates, the different owners want to ensure that no unwanted information flows between the various applications. If it were possible to install all applications at once before distributing the card to the public, many techniques would be available to check information flow.[3,4] Unfortunately, business users want asynchronous updates.

What remains out of reach is the combination of deploying new applications on a smart card once it's in the field and keeping the security certification. This calls for a costly manual review: developers must prove that all possible card evolutions are security-neutral so that their formal proof of compliance with Common Criteria is still valid and doesn't require a new certificate. The natural consequence is that no certified multimarket sector smart cards currently exist in the field, although both the GlobalPlatform and Java Card specifications support them.

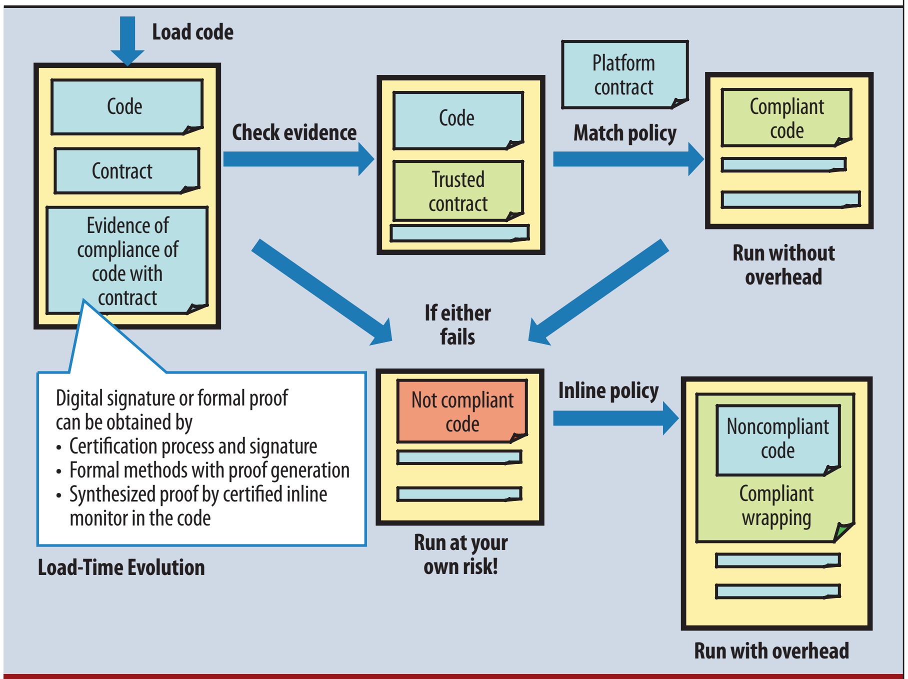

Figure 1. Security-by-Contract for load-time execution. We must follow specific steps to ensure that the code downloaded to a mobile or embedded device isn't harmful.

An emerging solution to this problem is the use of verification techniques to support LTE—that is, when the software is updated on a device already in the field. Sekar and colleagues suggested this basic idea when they introduced the notion of model-carrying code[7]: an application carries with itself a model to be verified at runtime. Unfortunately, this concept hasn't progressed because of significant limitations in the proposed model—for instance, it wasn't possible even to state policies such as "you should only connect to URLs starting with https://."

The Security-by-Contract framework[9] developed within the European S3MS project (www.s3ms.org) has shown concrete realization of the idea of complementing load-time and runtime checking for mobile phones running .NET and Java by using very expressive policies.[8] US researchers later ported the same approach to Google's Android platform.[10] The basic idea behind Security-by-Contract is that before loading software updates on the device, we extract the software's security-relevant behavior and compare it against our policy. If this behavior is acceptable, we load the software; if not, we can decide to use online monitoring techniques to make sure the software doesn't misbehave. This won't generate too much overhead, but in some cases it might not be feasible for resource-limited devices.

Figure 1 shows the basic intuitive workflow behind Security-by-Contract. In the simplest mode, the embedded or mobile device has just downloaded some new code that allegedly provides some desired functionality. How to check that it isn't harmful? We're at the beginning of the process in Figure 2; an untrusted code has been downloaded. We first extract the application contract `Claim` using `ContractExtractor` on the trusted part. At this point, we're interested in extracting security-relevant behaviors via data-, control-, and information-flow analysis[11] or from the application's manifest.[10] We then check whether this result matches the security policy `Policy` using `SimulationChecker`.[9] If the simulation succeeds, we can execute the code without further ado; otherwise, we use `Rewriter`, which gives the ready-to-be-executed result `SafeCode`.[8] Of course, `Rewriter` might introduce some overhead that, on embedded devices, might not be computationally acceptable. If the match with the policy is only partial, we can optimize the enforcement mechanism by using `Optimizer`, which gives the result `OptPolicy`—this contains only the bits of policy with which the contract wasn't compliant.

Of course, this approach assumes that everything can be done on the trusted side of the world—namely, on the embedded system itself. However, not all embedded systems have the same computational power: we can do some elementary checking of information flows on a smart card[11] and full automata verification on a mobile phone.[8] In many cases, we must trade off trustworthiness for computational power by deciding which operation the device can do by itself and on which operations it must rely for external help. At the extreme

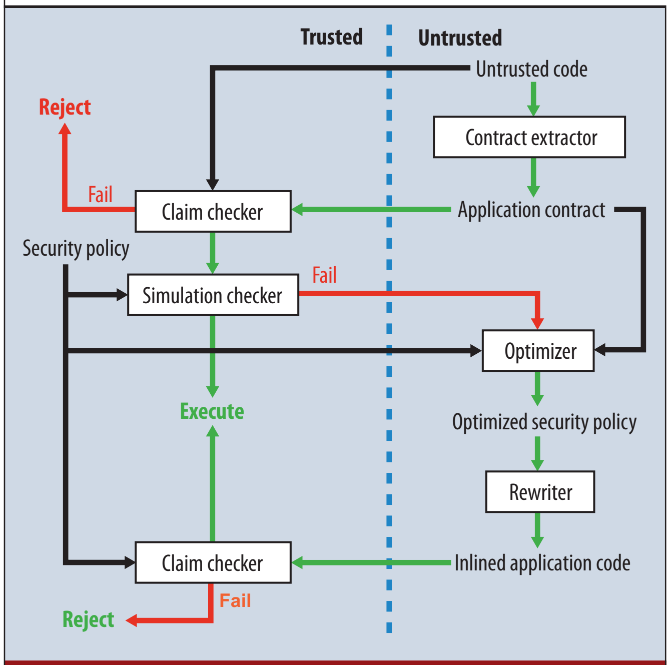

Figure 2. Load-time evaluation with trusted checking and untrusted computing. When the device doesn't have enough computational power, we shift costly computation to untrusted parties—checking their results is easier.

end of the spectrum, as Figure 2 shows, we move most of the components out of the trusted domain for the simple reason that the trusted domain—the embedded system—doesn't have enough computing power.

After running `ContractExtractor`, we check the application contract `Claim` against the application `Code` using `ClaimChecker`. If `Code` doesn't comply with `Claim`, then we reject `Code`. However, rejection might be too restrictive, so another similar option is to directly deploy the `Policy` object in charge of monitoring `Code` by using `Rewriter`, which gives the result `SafeCode`. By using load-time verification, we can thus overcome the limitation that certification imposes on the business model and achieve asynchronous evolution while guaranteeing security. Unfortunately, this approach might be too costly if we don't need to actually ensure that nothing bad ever happens with regard to safety and security, but we're satisfied that something good can possibly happen such as liveness, and that the most blatant violations aren't possible. In this setting, LTE verification might be effectively replaced by testing the embedded system for the desired behavior.

EVOLVING TESTS

Testing is the most widely used technique for evaluating a software-based system in its target environment: developers typically don't generate systems completely—a thorough model-based design process ultimately produces the system with all its ingredients in a formally verified chain of transformations—or formally checks systems in a way that completely verifies both system and environment.

DTE tests are fairly straightforward. They include retests for bug fixes, regression tests for modifications of existing functionalities, new tests for system extensions, and modified or new tests when environment changes affect the system itself. LTE and RTE tests are more difficult to define and perform. For LTE, when the system evolves offline, the necessary "testware"—test experts, environment, tools, and so on—is typically unavailable, so even lightweight tests for major system functionalities are hard to execute. One approach is to offer remote test capabilities[12] that enable testing an evolved system from a remote site automatically. This is an established method in other engineering disciplines such as automotive or industrial automation and could be adopted for software-intensive embedded systems as well.

An online setting is challenging because the tests aren't only remote but they also must evaluate the system in its target environment, which risks corrupting or damaging the system itself. However, testing must occur in a controlled environment to make the tests repeatable and stable in their results. The control typically includes setting the system's states and its environmental components, which generally isn't possible, necessitating a mixture of explicit control and passive observation (and deduction) instead. Such an approach helps minimize the impact on the running system. On the other hand, system functionalities must be elaborated as much as needed by stimulating the system in addition to its productive use: the system is stimulated with selected inputs, messages, operation calls, and so forth to activate system reactions that exhibit the functionality under consideration. The contradictory goals of minimal impact and explicit setting and stimuli are difficult to achieve, but approaches for built-in tests[13] provide some initial solutions.

Online tests require minimal functional interference with the running system and with other connected systems to avoid functional outages, and minimal resource consumption to avoid performance degradation. They allow systems to test themselves for constraints on their

- *environment*, whether it follows the environmental assumptions for which the system is built;
- *configurations*, whether the system is used in a setting for which it's constructed;
- *usage scenarios*, whether the system is used according to envisaged scenarios; and
- *their own reactions*, whether the reactions are outside of expected ranges.

Like LTE tests, RTE tests need to be online, but they also must be able to dynamically adapt to system changes during runtime. While LTE tests are rather static because possible system changes are predetermined, RTE tests must dynamically evolve whenever the system evolves. Hence, RTE tests require supervisory support to detect system changes during runtime and test adaptation support to enable changes to the tests accordingly.

Whenever tests identify faults, a supervisory system should also offer corrective means to adjust the system or its configuration where needed. Such a closed control loop between system, tests, and the evolutions thereof isn't easy to handle, especially because errors detected during testing can have their causes in the tests, in the system's requirements or specifications, or in the system itself. Before claiming the system to be faulty, we must rule out the other two options.

Using two different models for systems and tests might be a solution[14]: separate test models help us reason about systems and their tests on an abstract level, verify that tests are semantically correct with regard to the constraints defined by the system model, and derive executable tests by using an automated test execution platform. For evolving systems, the coordinated evolution of system models and test models is a challenge in itself: both must be synchronized, that is, consistent with regard to the constraints they impose. Approaches to model-based testing[15] provide some initial solutions for deriving tests on the fly when system models change. A delta approach, typically used in software debugging,[16] could also point a way forward.

In addition to functional tests that check a system's principal features and functionalities, nonfunctional tests can be enhanced for evolving systems, including tests for robustness to check that the system reacts safely in case of unexpected inputs or usage scenarios from the environment, for performance to check that it reacts as timely as needed, for scalability to check that it keeps its performance under an increasing load, and for security to check that it can withstand attacks. As an initial attempt to meet these challenges, we've developed an approach for automated performance and scalability tests and for automated test generation for embedded systems.[17] We're also developing a generic approach for the specification of reusable "X-in the loop" tests based on the well-established modeling and testing technologies Matlab/Simulink and TTCN-3.[18]

Embedded systems pose special challenges to system evolution: they're embedded in a changing environment, often interacting with evolving processes of human organizations, and thus must be verified because of their critical nature. Complicating the situation, the analyses and testing regimens used to verify them must evolve as well. Both software engineering research and industrial practice need to improve to address these problems. While admittedly underemphasized in software engineering education, system evolution is crucial, and the challenges discussed here will be addressed by improving on the initial results we presented. C

Acknowledgments

This work was in part sponsored by DARPA, under its Software Producibility Program; the US National Science Foundation, under award numbers CCR-0205575, CCR-0427071, and IIS-0705772; and the EU, under the projects EU-FP7-FET-IP-SecureChange and EU-FP7-IST-IP-MASTER. Any opinions, findings, and conclusions or recommendations are those of the authors and don't necessarily reflect the views of DARPA, the EU Commission, the US National Science Foundation, or the US government.

References

1. T. Levendovszky and G. Karsai, "An Active Pattern Infrastructure for Domain-Specific Languages," to appear in *Electronic Comm. EASST*, 2010; http://journal.ub.tu-berlin.de/index.php/eceasst/index.
2. B.H.C. Cheng et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," *Software Eng. for Self-Adaptive Systems*, LNCS 5525, Springer, 2009, pp. 1-26.
3. P. Bieber et al., "Checking Secure Interactions of Smart Card Applets: Extended Version," *J. Computer Security*, vol. 10, no. 4, 2002, pp. 369-398.
4. E. Hubbers, M. Oostdijk, and E. Poll, "From Finite State Machines to Provably Correct Java Card Applets," *Proc. IFIP TC11 18th Int'l Conf. Information Security* (SEC 03), Kluwer Publishers, 2003, pp. 465-470.
5. B. Chen et al., "Analyzing Medical Processes," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), ACM Press, 2008, pp. 623-632.
6. J.M. Cobleigh, G.S. Avrunin, and L.A. Clarke, "Breaking Up Is Hard to Do: An Evaluation of Automated Assume-Guarantee Reasoning," *ACM Trans. Software Eng. Methodologies*, vol. 17, no. 2, 2008, pp. 1-52.
7. R. Sekar et al., "Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications," *ACM Symp. Operating Systems Principles* (SOSP 03), ACM Press, 2003, pp. 15-28.
8. L. Desmet et al., "Security-by-Contract on the .NET Platform," *Information Security Technical Report*, vol. 13, no. 1, 2008, pp. 25-32.
9. N. Dragoni et al., "Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code," *Proc. 4th European PKI Workshop* (EuroPKI 07), LNCS 4582, Springer, 2007, pp. 297-312.
10. W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," *Proc. 16th ACM Conf. Computer and Communications Security* (CCS 09), ACM Press, 2009, pp. 235-245.
11. D. Ghindici, G. Grimaud, and I. Simplot-Ryl, "An Information Flow Verifier for Small Embedded Systems," *Proc. Int'l Workshop Information Security Theory and Practices* (WISTP 07), LNCS 4462, Springer, 2007, pp. 189-201.

12. P.H. Deussen, "Supervision of Autonomic Systems," *Int'l Trans. Systems Science and Applications*, vol. 2, no. 1, 2006, pp. 105-110.
13. H.-G. Gross, I. Schieferdecker, and G. Din, "Model-Based Built-In Tests," *Electronic Notes in Theoretical Computer Science*, vol. 111, 2005, pp. 161-182.
14. P. Baker et al., *Model-Driven Testing: Using the UML Testing Profile*, Springer, 2007.
15. L. Frantzen, J. Tretmans, and T.A.C. Willemse, "A Symbolic Framework for Model-Based Testing," *Formal Approaches to Software Testing and Runtime Verification*, LNCS 4262, Springer, 2006, pp. 40-54.
16. A. Zeller, "Debugging Debugging: ACM Sigsoft Impact Paper Award Keynote," *Proc. 7th Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.* (ESEC-FSE 07), ACM Press, 2009, pp. 263-264.
17. J. Zander-Nowicka, X. Xiong, and I. Schieferdecker, "Systematic Test Data Generation for Embedded Software," *Proc. Software Eng. Research and Practice* (SERP 08), vol. 1, CSREA Press, 2008, pp. 164-170.
18. J. Grossmann, D.A. Serbanescu, and I. Schieferdecker, "Testing Embedded Real Time Systems with TTCN-3," *Proc. 2009 Int'l Conf. Software Testing Verification and Validation* (ICST 09), IEEE CS Press, 2009, pp. 81-90.

Gabor Karsai *is a professor of electrical engineering and computer science at Vanderbilt University and a senior research scientist in its Institute for Software-Integrated Systems. His research is in model-integrated computing. Karsai received a PhD in electrical engineering from Vanderbilt University. He's a member of the IEEE Computer Society. Contact him at gabor.karsai@vanderbilt.edu.*

Fabio Massacci *is a professor of computer security at the University of Trento, Italy. His research interests are in security requirements engineering and security verification for mobile systems. Massacci received a PhD in computer science and engineering from Sapienza University of Rome, Italy. He is a member of the ACM, IEEE, and ISACA. Contact him at fabio.massaci@unitn.it.*

Leon Osterweil *is a professor of computer science at the University of Massachusetts Amherst and codirector of its Laboratory for Advanced Software Engineering Research and the Electronic Enterprise Institute. His research centers on software analysis and testing, software tool integration, and software processes and process programming. Osterweil received a PhD in mathematics from the University of Maryland. He is a fellow of the ACM. Contact him at ljo@cs.umass.edu.*

Ina Schieferdecker *heads the Competence Center on Modeling and Testing of System and Service Solutions at Fraunhofer FOKUS, Berlin, and is also a professor of design and testing of communication-based systems at Technical University Berlin. Her research interests include model-driven engineering, software quality assurance, conformance, interoperability, and certification. Schieferdecker received a PhD in electrical engineering from Technical University Berlin. She is a member of IEEE, the ACM, the German Academy of Science and Engineering (Acatech), Gesellschaft für Informatik, and ASQF. Contact her at ina.schieferdecker@fokus.fraunhofer.de.*