



{EssentialSet}

Evolving Critical Systems

Edited by Mike Hinchey, Lorcan Coyle,
Bashar Nuseibeh, and José Luis Fiadeiro



Essential Articles on Software Engineering



Easily browse
this EssentialSet
using buttons
or bookmarks.

Copyright and Reprint Permissions: Educational or personal use of this material is permitted without fee provided such copies 1) are not made for profit or in lieu of purchasing copies for classes, and that this notice and a full citation to the original work appear on the first page of the copy and 2) do not imply IEEE endorsement of any third-party products or services. Permission to reprint/republish this material for commercial, advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org.

IEEE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL IEEE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF IEEE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Copyright © 2011 IEEE. All rights reserved.

User Note:

Some versions of Adobe will prompt the reader to interact with the EssentialSet using the "Forms" functionality. Please disregard such a prompt, minimizing the pink bar and selecting "do not show again" when asked.

TABLE OF CONTENTS

Introduction

by Mike Hinchey, Lorcan Coyle, Bashar Nuseibeh, and José Luiz Fiadeiro

Guest Editors' Introduction: Evolving Critical Systems

by Lorcan Coyle, Mike Hinchey, Bashar Nuseibeh, and José Luiz Fiadeiro

Provides an overview of Evolving Critical Systems and introduces the other papers in this collection.

Evolving Embedded Systems

by Gabor Karsai, Fabio Massacci, Leon Osterweil, and Ina Schieferdecker

In this article, the authors discuss four selected aspects of evolution in embedded systems: the different time-scales of evolution, the coevolution of processes and systems, checking evolution at deployment time, and the evolution of tests used to check system correctness.

Evolving Software Architecture Descriptions of Critical Systems

by Tom Mens, Jeff Magee, and Bernhard Rumpe

To manage the complexity of developing, maintaining, and evolving a critical software-intensive system, its architecture description must be accurately and traceably linked to its implementation.

Evolution in Relation to Risk and Trust Management

by Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen

A methodology within risk and trust management in general, and risk and trust assessment in particular, isn't well equipped to address trust issues in evolution.

Why Critical Systems Need Help to Evolve

by Bernard Cohen and Philip Boxer

Classical engineering fails to model all the ways in which a critical sociotechnical system fits into a larger system. A study of orthotics clinics used projective analysis to better understand the clinics' role in a healthcare system and to identify risks to the clinics' evolution.

Simplicity as a Driver for Agile Innovation

by Tiziana Margaria and Bernhard Steffen

Software and hardware vendors long avoided inter-operation for fear of opting out of their own product lines. Yet decisive change came to the automobile industry from a holistic evolution and maturation on many fronts.

Recommended References

About the Editors



INTRODUCTION

There are few areas of modern life in which software is not an important (though often invisible) component. The software in our lives is increasingly complex; its interaction with the real world means that its requirements are in a state of constant change. Many non-software products and services, from healthcare to transport, education to business, depend on reliable, high-quality software.

Software systems frequently need to be modified in response to changes in system requirements and in their operational environment [1]. Such modification may involve the addition of new functionality, the adjustment of existing functions, or the wholesale replacement of entire sub-systems. All such change is fraught with uncertainty—software projects involving change frequently fail to meet requirements, run over time and budget, or are abandoned [2].

As the ubiquity and complexity of software increase, a requirement has emerged for **critical software that can successfully evolve** without loss of quality—software that is engineered from the start to be easily

changed, extended, and reconfigured, while retaining its security, its performance, and its reliability and predictability.

Software Evolution

The problem of how to modify software easily without losing quality was widely understood and discussed at the NATO Software Engineering Conference in 1968 [3]. Lehman et al.'s early work on the continuing change process of the IBM OS360-370 operating systems and the work that followed from that led to a large body of research into software evolution and the formulation of eight “Laws of Evolution” [4].

There are three types of evolution [1]:

1. *corrective maintenance*, used to overcome processing failure, performance failure, and implementation failure;

2. *adaptive maintenance*, which would overcome change in data environment (e.g., restructuring of a database) and change in processing environment (new hardware, etc.); and

3. *perfective maintenance*, which would improve design, which might overcome processing inefficiency and enhance both performance and the system's maintainability.

More mature software, where many (or all) of the key developers are no longer in place, is seen as being harder to evolve than newer software supported by its original developers [2].

As software evolves in terms of functionality, it often degrades in terms of reliability. While it is normal to experience failures after deployment, and the goal of much of software maintenance is to remove these failures, experience has shown that evolution for new functionality and evolution for maintenance can both result in “spikes” of failure. Over time, a traditional system degrades as it evolves and more, rather than fewer, failures are experienced [2, 4].

Dynamic evolution (sometimes called run-time or automatic evolution) is a special case whereby certain critical systems may need to change during run-time, e.g., by hot-swapping existing components

or by integrating newly developed components without first stopping the system [5]. If this has not been planned ahead explicitly in the system, the underlying platform has to provide a means to effectuate software changes dynamically. In terms of the software evolving itself automatically, there are a number of challenges beyond those faced when a human drives the process. Ubiquitous computing systems or autonomic systems are often typified as consisting of large numbers of distributed autonomic, often resource-constrained embedded systems. Designers cannot fully predict how a system will behave and how it will interconnect with a continuously changing environment. Therefore, software must adapt and react to change dynamically, even if such change is unanticipated.

Critical Systems

Critical systems are systems where failure or malfunction will lead to significant negative consequences [6]. These systems may have strict requirements for security and safety, to protect the user or others [7]. Alternatively, these systems may be critical to the organization's

mission, product base, profitability, or competitive advantage. For example, an online retailer may be able to tolerate the unavailability of their warehousing system for several hours in a day, since most customers will still receive their orders when promised. However, unavailability of the website and ordering system for several hours may result in the permanent loss of business to a competitor. A brief categorization of types of critical systems is shown in Table 1.

Evolving Critical Systems

Evolving systems (Lehman called these "E-type" systems [4]) may:

- have evolved from legacy code and legacy systems;
- result from a combination of existing component-based systems, possibly over significant periods of time;
- be the result of the extension of an existing system to include new functional requirements;
- evolve as the result of a need to improve their quality of service, such as performance, reliability, usability, or other quality requirements;
- evolve as a result of an intentional change to exploit new

Table 1: Types of Critical Systems: Many systems have overlapping aspects of criticality, e.g., a system might be both safety-critical and business-critical.

Type of Critical	Implication for Failure
Safety-Critical	May lead to loss of life, serious personal injury, or damage to the natural environment.
Mission-Critical	May lead to an inability to complete the overall system or project objectives; e.g., loss of critical infrastructure or data.
Business-Critical	May lead to significant tangible or intangible economic costs; e.g., loss of business or damage to reputation.
Security-Critical	May lead to loss of sensitive data through theft or accidental loss.

technologies and techniques, e.g., service-oriented architectures, or a move toward multi-core-based implementations;

- adapt and evolve at run-time in order to react to changes in the environment or to meet new requirements or constraints.

Most large and complex software systems are evolving systems. The alternative to system evolution is total replacement, often not feasible for cost and other reasons.

Conclusions

Critical computer-based systems are ever more important in our lives, but as they age they will be in need of increasing amounts of effort to evolve them to remain useful [4]. Much work needs to be done to help ease this burden.

Evolving Critical Systems is a research area that tackles some of the challenges and important research questions that face us. Given that software evolution can be seen as a

compromise between cost and risk, the most pressing question to ask is: Which processes, techniques, and tools are most cost-effective for evolving critical systems?

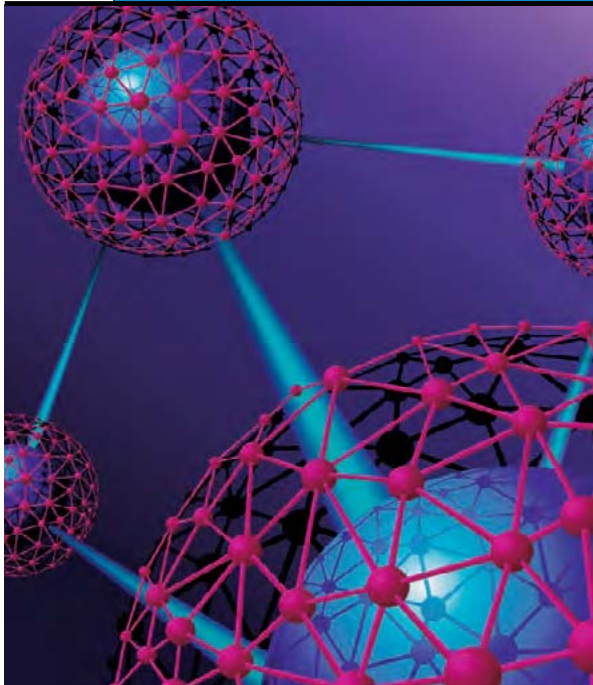
Mike Hinchey
Lorcan Coyle
Bashar Nuseibeh
José Luiz Fiadeiro
June 2011

Guest Editors' Introduction: Evolving Critical Systems

by Lorcan Coyle, Mike Hinchey, Bashar Nuseibeh,
and José Luiz Fiadeiro

Computer, vol. 43, no. 5, May 2010, pp. 28–33

DOI bookmark: <http://doi.ieeecomputersociety.org/10.1109/MC.2010.139>



GUEST EDITORS' INTRODUCTION: EVOLVING CRITICAL SYSTEMS

Lorcan Coyle, Mike Hinchey, and Bashar Nuseibeh, *Lero—the Irish Software Engineering Research Centre*
 José Luiz Fiadeiro, *University of Leicester*

This special issue brings together key software engineering researchers and practitioners who both influence their organizations and evaluate the emerging practice of developing these new systems.

We believe that the software engineering community must concentrate efforts on the techniques, methodologies, and tools needed to design, implement, and maintain critical software systems that evolve successfully. This special issue summarizes many of the topics discussed and embodies what we believe to be some of the most important research challenges for evolving critical software systems—without incurring prohibitive costs.

Several widespread changes in software engineering highlight the importance of evolving critical systems (ECS). We've identified the following five game changers:

- **Software ubiquity.** More software is being deployed in more consumer devices, which means failures are more likely to affect ordinary people.
- **Software criticality.** As software embeds itself deeper into the fabric of society, single software failures

have greater potential to affect more people. This increases the potential for software to be considered critical even when it isn't complex.

- **People-in-the-loop.** As software is deployed to control systems in which human actors participate, the issue of human interactions with software becomes more important.
- **Entanglement.** Software dependencies have become more complex, and much real-world software is entangled with software developed by third-party providers.
- **Increased evolution tempo.** The tempo of evolution will continue increasing as users expect more from software. The software market is often unforgiving when even small changes can't be done cheaply and quickly.

Taken together, these changes characterize evolving critical systems and frame the research agenda in this emerging area (www.lero.ie/ecs/whitepaper).

ECS MANIFESTO

ECS research challenges add to the broad software engineering research agenda, with more of a focus on predictability, quality, and the ability to change. Table 1 characterizes the criticality of software engineering research challenges for ECS.

Table 1. Critical system types.

Type	Implications for failure
Safety-critical	Can lead to loss of life, serious personal injury, or damage to the natural environment
Mission-critical	Can lead to an inability to complete the overall system or project objectives, such as loss of critical infrastructure or data
Business-critical	Can lead to significant tangible or intangible economic costs such as loss of business or damage to reputation
Security-critical	Can lead to loss of sensitive data through theft or accidental loss

The fundamental research question underlying ECS research is this: how do we design, implement, and maintain critical software systems that are highly reliable while retaining this reliability as they evolve, without incurring prohibitive costs? Several demands must be met before ECS's ideals can be realized.

The changing development environment, for example, proves that we must maintain the quality of critical software despite constant change in its teams, processes, methods, and toolkits. Likewise, we must improve our existing software design methodologies so that they facilitate the support and maintenance of ECS—how can we use agile development methodologies to evolve critical software?

We must also specify what we want to achieve during an evolution cycle and confirm that we've achieved the intended result (verification) and only the result intended (validation). We must thus elicit and represent requirements for change such that we ensure the changes take place correctly. Furthermore, we must develop techniques for better estimating specific evolution activities a priori, only attempting software change when we know for certain that evolution will be successful and that benefits will outweigh costs. For example, some systems shouldn't evolve at all because the cost and risk of performing evolution successfully will exceed the system's value by orders of magnitude. To prevent cost and time overruns, we must work toward developing objective criteria to help us decide whether a given system is in this class.

All these requirements demand strategies to make model-driven, automatic evolution a better alternative to manual change. In cases where it isn't appropriate to mechanize change, we must develop heuristics for determining when such an approach is viable. When humans must perform the change, we need to develop support tools that make this a less risky enterprise.

We also need improved tools for traceability that keep various software artifacts—such as documentation and source code—in sync throughout the evolution cycle. Where regulatory compliance is required, these tools must ensure that evolution results in compliant software.

Finally, during runtime evolution, we must ensure that developers adhere to runtime policies. We do so by developing techniques that can monitor and model changing

requirements in dynamic environments, especially automatic and adaptive software environments. We must also develop strategies for evolution that tolerate uncertainty in the operational environment, which changes deterministically, nondeterministically, or stochastically. We must then ensure that software never evolves into a state of unstable behavior.

Given the tensions between the need for software change and the danger implicit in changing critical software, the most pressing question for practitioners is which processes, techniques, and tools can most cost-effectively evolve critical systems. We believe a concerted focus from the research community to overcome these challenges will be needed for ECS to have an impact on software engineering.

IN THIS ISSUE

This special issue contains contributions from leading participants in the field. In "Evolving Embedded Systems," Gabor Karsai and coauthors discuss the importance of considering the interplay among requirements, processes, deployments, and tests' evolution when developing embedded systems. They also address challenges relating to the different evolutionary time scales—whether the system is evolved at design time, load time, or runtime.

In addition, the authors examine the importance of process and system co-evolution, especially with regard to embedded systems, discussing how verification can be used to check the correctness of load-time evolution. They conclude by addressing the difficulties inherent in testing software that evolves at load time or runtime, while also relating it to the use of online testing, built-in testing for load-time evolution, and the need to evolve the tests themselves, especially for runtime evolution.

In "Evolving Software Architecture Descriptions of Critical Systems," Tom Mens and coauthors discuss the use of architectural descriptions to describe software-intensive systems and how they can be used to handle increasing complexity to mitigate the risks incurred in constructing and evolving these systems. They also assess the use of model-transformation approaches to evolve models of software architectures and the co-evolution of architecture descriptions, software design, and implementation. They call for architectural change to be considered a first-class construct that ensures

➔ ASSESSING CRITICALITY IN AUTOMOTIVE SYSTEMS

Dieter Lienert and Stefan Kriso, Robert Bosch GmbH

Consider the problem of assessing criticality in automotive systems. One important aspect of most such systems is functional safety. This is addressed in ISO 26262, a forthcoming standard for functional safety of electrical and electronic (E/E) systems in road vehicles. Currently, a draft international standard (DIS), it's expected to be approved by the International Organization for Standardization by mid-2011. To reduce unpredictable product liability risks, all road vehicles brought to market after publication of the final standard must conform to ISO 26262. This means that E/E system developers must consider all development process and product properties requirements mandated by the standard from the beginning.

To specify the criticality of E/E system malfunctions, ISO 26262 defines the Automotive Safety Integrity Level (ASIL), which ranges from A (lowest) to D (highest); a QM value indicates that a malfunction isn't safety-related. As Figure A shows, developers must estimate three parameters to determine ASIL:

- **Exposure (E).** This factor defines the probability of a system being in an operational situation that can be hazardous if coincident with a failure mode; rated on a scale from E0 (incredible) to E4 (high probability).
- **Controllability (C).** This factor assesses the potential of avoiding specific harm or damage through the timely reactions of the persons involved; rated on a scale from C0 (controllable in general) to C3 (difficult to control or uncontrollable).
- **Severity (S).** This parameter measures the potential extent of harm to an individual in a specific situation; rated on a scale from S0 (no injuries) to S3 (life-threatening or fatal injuries).

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Figure A. Automotive Safety Integrity Level determination. Electrical/electronic system developers must estimate three parameters—exposure (E), controllability (C), and severity (S)—to determine a malfunction's ASIL (Source: ISO/DIS 26262-3, Table 4.)

Various interpretations of the categories' meaning have been defined for E, C, and S. To avoid different ASIL classifications for the same malfunction, the automotive industry must establish common criteria.

An appropriate system design makes reducing ASIL classifications for some elements, a process known as ASIL decomposition. In this case, the elements' independence after decomposition must be assured—"lower-quality" parts mustn't affect the operation of "critical" parts that assure safety.

For example, software malfunctions within the electronic actuator pedal (EGAS) system may lead to the ASIL B, as the top of Figure B shows. This is decomposed into QM for the function level and B for the monitoring level (as the bottom of the figure shows), which assures safety by switching off the power stages if the level malfunctions.

In the case of E/E systems, automotive engineers are thus guided by a well-established set of different safety-integrity levels rather than a general notion of criticality. However, technical challenges arise from the problem of coping with different ASILs within one application. And this situation will probably occur more frequently given the growing trend to build functionally cooperating networks of originally separate systems.

More challenges arise if safety requirements conflict with other types of criticality, such as system reliability and availability (quality of service). For example, a fail-safe solution like switching off the power stages in the EGAS might lead to customer dissatisfaction if it occurs too often. ■

Dieter Lienert is the head of a group for the engineering of software-intensive systems at Robert Bosch GmbH. Contact him at dieter.lienert@de.bosch.com.

Stefan Kriso is a senior project manager at Robert Bosch GmbH. Contact him at stefan.kriso@de.bosch.com.

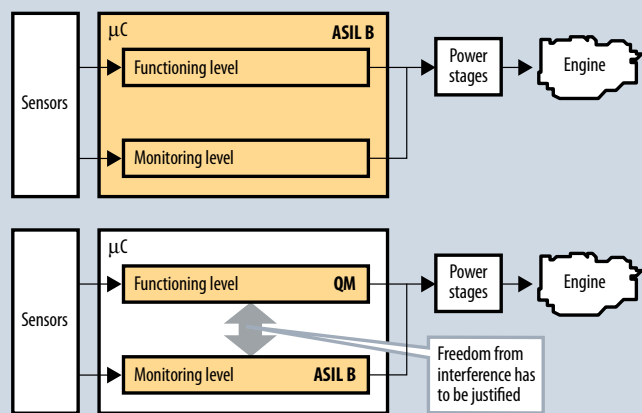


Figure B. Example of ASIL decomposition for the electronic actuator pedal (EGAS) system. Top: Software malfunctions in the microcontroller (µC) are classified as ASIL B. Bottom: the monitoring level is classified as ASIL B, while the function level can be classified as QM if freedom from interference between the levels can be justified.

➔ SMART-CARD CERTIFICATION

Bouthaina Chetali, Security Labs—Gemalto

Smart-card certification's evolution can't be tackled as a whole, but only as an update evolving into a new service or capability. The software's update focuses on the context of product surveillance and maintenance, two processes clearly defined by the certification scheme. To update a certified product and maintain the certificate, the developer must provide evidence that this update has no security impact.

Open smart cards

Requirements such as these could burden a business model, where software updates must be frequent to keep pace with the rapid evolution of specifications. However, the most interesting evolution deals with new services first, then with new code applications. For that, the trend is the certification of "open" smart cards.

Smart cards that have been certified as "open" could be used to load any kind of applications through binary code when in the field, while keeping its certificate intact. An open card has no applications onboard and essentially becomes an "operating system"—such as a Java Card platform.

This sophisticated application manages card resources securely by, for example, loading, installing, deleting, and delivering updates. The openness relies on the loading mechanism and its isolation properties, and these security mechanisms have been evaluated during certification to provide the necessary guarantees that the product can load any code and that two different applications can remain secure and protected from each other.

Certification rules

To maintain stability, certification includes rules called hypotheses that applications must respect before being loaded onto the card. These rules rely on blocking attack paths. One well-known rule advises that code be checked by a bytecode verifier, which means the application must be bytecode-verified before loading, if the onboard verification is not a feature of the card.

The rules set represents the card issuer policy, but if the applications originate elsewhere, in different market sectors that don't have the same historic level of security requirements, such as banking and mobile communication, a common agreement on policy rules could present a complex task. Therefore, it seems obvious that if the product is protected against any kind of application that could be loaded on the card, the underlying software must include a large set of protective countermeasures.

For a constrained-resources device such as a smart card, adding ever more software countermeasures or onboard verification leads to performance issues during execution and also consumes card-memory space. Both are crucial for the end user's satisfaction: the first is response time to the request, the second the number of applications that can load onto the product.

Essentially, then, the challenge resides on performance rates, and a trade-off must be made between security and speed. So the notion of evolution is not the same for each sector: the lifetime of a banking card differs from that of a SIM card. This is why taking certification requirements and evolution requirements for cards into account when hosting applications from different market sectors poses a daunting challenge for the smart-card industry. ■

Bouthaina Chetali manages the Formal Methods Group at Security Labs—Gemalto. Contact her at bouthaina.chetali@gemalto.com.

co-evolution between the architecture descriptions and their implementation.

In "Evolution in Relation to Risk and Trust Management," Mass Soldal Lund and coauthors argue that risk and trust management methodologies in general, and assessment in particular, aren't well-equipped to deal with evolution. They go on to explore risk assessment from the perspectives of maintenance, in which old risks must be updated to take into account new risks introduced by changes before and after, in which the assessor must be aware of current risks, future risks, and risks introduced by the change process itself; and continuous evolution, which involves identifying and assessing how risks evolve. The authors also explore using each of these two perspectives from the specific viewpoint of the risk that trust relations impose on a system. They assert that the evolution of trust is much more challenging, given the highly dynamic nature of trust relations.

Finally, in "Why Critical Systems Need Help to Evolve," Bernie Cohen and Philip Boxer analyze the difficulties in evolving complex sociotechnical systems by using the provision of orthotic services in the UK as an exemplar. They use three *cuts* to address the risks of ECS. The Car-

tesian cut presents a mismatch between the model that defines a system and the reality of its interaction with stakeholders; the Heisenberg cut presents a mismatch between what behaviors can and can't be predicted by its users, independently of their use of it; and the Endo-exo cut reflects the difference between what can and can't be directly known by clients about their own needs. By enabling the members of and stakeholders in a sociotechnical system to analyze and project the experience of their own participation, the authors gain an understanding of how the orthotic service makes the three cuts and identify changes that could improve it. Significantly, these changes were ultimately rejected by the UK's National Health Service. The authors suggest that this stemmed from a failure to understand the ecosystem in which the service was embedded and the wider implications of these changes beyond the orthotics services.

In addition to this introduction, we've included three shorter practitioner contributions from Robert Bosch GmbH, Security Labs—Gemalto, and the Directorate General for Informatics in the European Commission.

➔ CRITICAL SYSTEMS ENABLING EUROPEAN INTEGRATION

Declan Deasy and Franck Noël, Directorate General for Informatics, European Commission

Sixty years ago, European leaders founded what has become the European Union. Starting with the European Coal and Steel Community in 1951, efforts to promote policy cooperation on the continent have dramatically expanded from energy to embrace agriculture, regional development, information technology, research, and more. The EU has also widened its geographical scope. Starting as a community of six member states, it has become a union of 27 members with 500 million people and a single currency, the euro.

The EU is a striking example of the positive transformation of mentalities and culture. European political integration has been accompanied by the creation of a new institutional architecture to address the complexity of new policies, decision-making processes, partners, and challenges in an ever-changing world. The rhythm and path of European integration have largely been influenced by geopolitical, social, and historical factors. However, technological evolution has also played a major role. In the past 60 years, information and communication technologies (ICT) have emerged from research labs to become essential elements of Europeans' daily lives at work, home, and play. Without ICT, the EU would have remained an abstract, paper-based construction.

Many such systems are critical to the EU's mission, political objectives, and reputation; their failure would lead to significant negative political consequences. As such, they have strict requirements for security and safety to protect their information assets and users. For example, customs unions rely on various control systems to manage the flow of goods and people across member-state borders. Consider the Schengen Area, which allows free movement of EU citizens without passports. This wouldn't be possible without the accompanying information system. European laws are easily accessible online to citizens and lawyers via the EUR-LEX system. The Internal Market Information system underpins the Single Market, in particular the Services Directive. The EU Structural Funds program, whose regional financing development is managed electronically, accelerates procedures, increases the reliability and transparency of transactions, and ultimately contributes to better use of taxpayers' money. In the environmental area, the Community Independent Transaction Log is at the core of the European carbon emission trading scheme implemented as part of the Kyoto Protocol.

The story doesn't end there. A new era of collaboration among public actors is quickly emerging to tackle global challenges such as the current economic crisis and climate change. This effort requires the support of a new class of cross-border and cross-sector critical information systems capable of evolving over time to accommodate disparate political and user needs. In the Ministerial Declaration on eGovernment, unanimously approved in Malmö, Sweden, in November 2009, EU public administration ministers recognized that "eGovernment has not only become mainstream in national policies but has also reached beyond national boundaries to become an important enabler to deliver European-wide policy goals across different sectors." They pledged to jointly strive to achieve the following policy priorities:

- "Citizens and businesses are empowered by eGovernment services designed around users' needs and developed in

collaboration with third parties, as well as increased access to public information, strengthened transparency and effective means for involvement of stakeholders in the policy process";

- "Mobility in the Single Market is reinforced by seamless eGovernment services for the setting up and running of a business and for studying, working, residing, and retiring anywhere in the European Union"; and
- "Efficiency and effectiveness is enabled by a constant effort to use eGovernment to reduce the administrative burden, improve organizational processes and promote a sustainable low-carbon economy."

Aligned with this declaration, the European Commission is preparing to modernize its portfolio of mission-critical information systems to deliver and operate smart e-government services that are innovative and built from a user-centric viewpoint, enabling their participation in the underlying processes (empowerment); and streamline administrative processes in a learning organization to improve effectiveness, efficiency, and transparency, and to share and value intellectual assets through appropriate knowledge-management approaches. This new generation of critical systems will be built on three principles: harmonization and convergence of business processes, reusability and interoperability of information systems or systems components, and sharing services at the infrastructure level. These three layers of the EC's IT Enterprise Architecture Framework will be enabled by an organization possessing the necessary IT governance arrangements and project management methodologies supporting top-caliber staff whose skills and knowledge will be continuously improved through collaboration in multidisciplinary teams.

The new mission-critical systems will be developed under the auspices of the EU's Interoperability Solutions for European Public Administrations (ISA) program, which came into being on 1 January 2010. ISA's focus is on back-office solutions to support the interaction between European public administrations and the implementation of EU policies and activities. It underlines the key role that standards and interoperability at all levels—legal, organizational, semantic, and technical—will play in ensuring these new systems contribute to European integration. Developments will conform to the ISA's European Interoperability Strategy, will be based on its European Interoperability Framework, and will respect the related architectural guidelines. The program thus defines the architecture for the next generation of evolving critical information systems upon which the EU will rely to implement the Europe 2020 vision articulated by EC President José Manuel Barroso.

E-government is now mainstream; in the EU and elsewhere it will be the catalyst in transforming public administrations over the next decade. The challenge for today's public-sector CIOs is to build evolving critical information systems that offer more online public services, streamline administrative procedures and cut red tape, and implement innovative service delivery mechanisms. The emerging evolving critical systems research domain will contribute to meeting this challenge. ■

Declan Deasy is Director of Information Systems at the European Commission in Brussels. Contact him at declan.deasy@ec.europa.eu.

Franck Noël is Deputy Head of Unit in the European Commission in Brussels. Contact him at franck.noel@ec.europa.eu.

Dieter Lienert and Stefan Kriso describe the emerging functional safety standard for electrical and electronic automotive systems (ISO 26262) and discuss the challenges in assessing criticality in automotive systems.

Boutheina Chetali points out that the smart-card industry faces a significant challenge in managing both certification and evolution requirements.

Finally, Franck Noël and Declan Deasy discuss how European integration within the EU has, in many cases, been enabled and accelerated by the development and evolution of their technical infrastructure. ■

Acknowledgments

This special issue resulted from a workshop held in Schloss Dagstuhl, Germany, in December 2009. The gathering brought together key software engineering researchers and practitioners in positions to influence their organizations' research direction and discuss the emerging theme of ECS.

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre (www.lero.ie). We thank Schloss Dagstuhl for hosting the perspectives workshop, and all the workshop participants whose discussions led to this special issue.

Lorcan Coyle is a research fellow at Lero in the University of Limerick. His research interests include autonomic computing, context awareness, pervasive computing, and machine learning. Coyle received a PhD in computer science from Trinity College Dublin. He is a member of the Institution of Engineers of Ireland. Contact him at lorcan.coyle@lero.ie.

Mike Hinchey is scientific director of Lero and a professor of software engineering at the University of Limerick. His research interests include self-managing software and formal methods for system development. Hinchey received a PhD in computer science from the University of Cambridge. He is a senior member of IEEE and currently chairs the IFIP Technical Assembly. Contact him at mike.hinchey@lero.ie.

Bashar Nuseibeh is a professor of software engineering and chief scientist for Lero. He is also a professor of computing at The Open University, UK, and a visiting professor at Imperial College London and the National Institute of Informatics, Japan. His research interests include requirements engineering and design, security and privacy, and technology transfer. Nuseibeh holds a PhD in software engineering from Imperial College London and is editor in chief of IEEE Trans. Software Eng. and editor emeritus of the Automated Software Eng. J. He is also a fellow of the British Computer Society (BCS) and the Institution of Engineering and Technology, and an Automated Software Engineering Fellow. Contact him at bashar.nuseibeh@lero.ie.

José Luiz Fiadeiro is a professor of software science and engineering in the Department of Computer Science at the University of Leicester. His research interests lie in the mathematical foundations of software system modeling, including software architecture, coordination models and languages, parallel and distributed system design, and service-oriented computing. Fiadeiro received a PhD in mathematics from the Technical University of Lisbon. He is a fellow of the BCS. Contact him at jose@mcs.le.ac.uk.



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.

Where Are We Headed?

Rich Internet applications are a heterogeneous family of solutions, characterized by a common goal of adding new capabilities to the conventional hypertext-based Web. RIAs combine the Web's lightweight distribution architecture with desktop applications' interface interactivity and computation power, and the resulting combination improves all the elements of a Web application.

Read the latest issue of *IEEE Internet Computing* now!

www.computer.org/internet

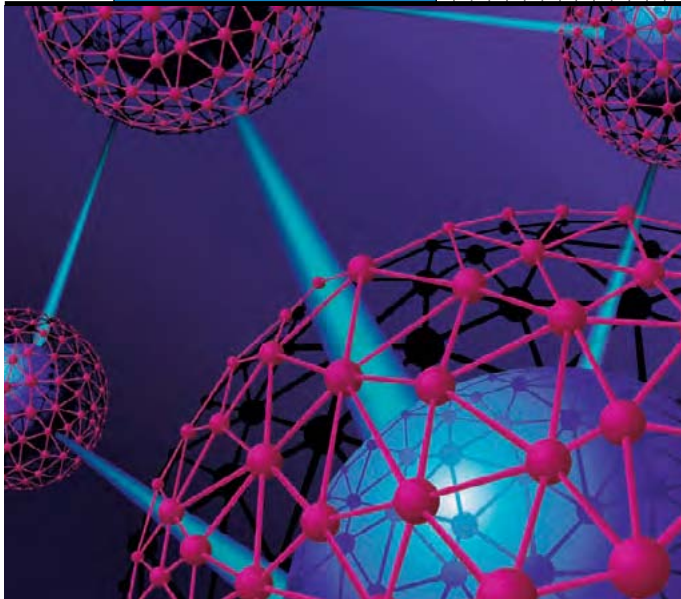


Evolving Embedded Systems

by Gabor Karsai, Fabio Massacci, Leon Osterweil,
and Ina Schieferdecker

Computer, vol. 43, no. 5, May 2010, pp. 34–40

DOI bookmark: <http://doi.ieeecomputersociety.org/10.1109/MC.2010.135>



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 re-verification 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 cor-

rectness. 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



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.

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.

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 language⁵ 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-reasoning⁶ and model-carrying code⁷ (or its modern variants⁸) 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 up-

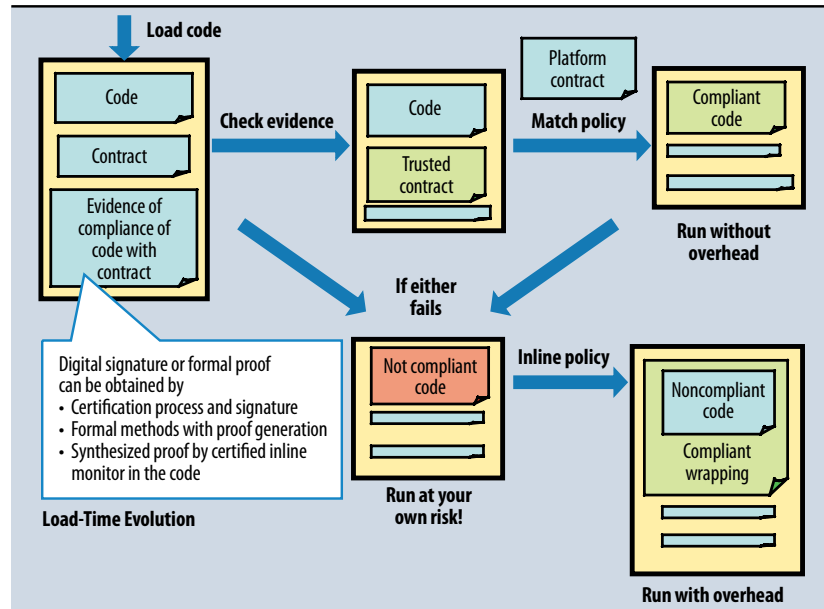
dates, the different owners want to ensure that no unwanted information flows between the various applications. If it were possible to install all applications at once before distributing the card to the public, many techniques would be available to check information flow.^{3,4} Unfortunately, business 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.

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 code⁷: 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 framework⁹ 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 analysis¹¹ 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 card¹¹ 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

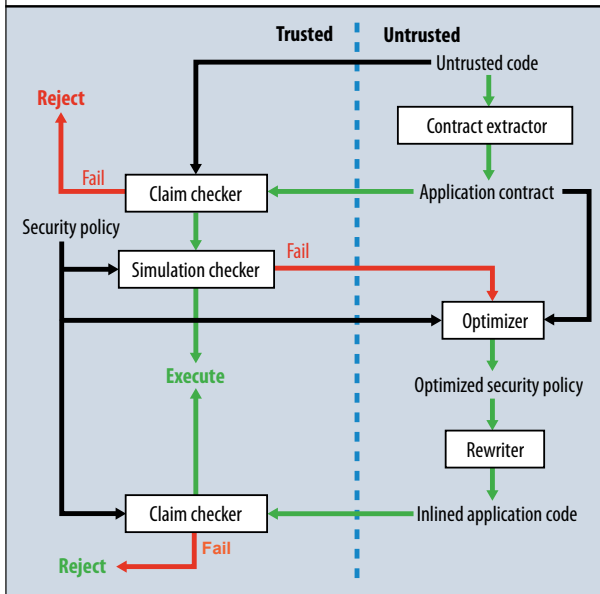


Figure 2. Load-time evaluation with trusted checking and untrusted computing. When the device doesn't have enough computational power, we shift costly computation to untrusted parties—checking their results is easier.

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 capabilities¹² 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 tests¹³ 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 solution¹⁴: 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 testing¹⁵ provide some initial solutions for deriving tests on the fly when system models change. A delta approach, typically used in software debugging¹⁶ 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. ■

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/leceasst/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.massacci@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.



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.

“All writers are vain,
selfish and lazy.”

—George Orwell, “Why I Write” (1947)

(except ours!)



The IEEE Computer Society Press is currently seeking authors. The CS Press publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. It offers authors the prestige of the IEEE Computer Society imprint, combined with the worldwide sales and marketing power of our partner, the international scientific and technical publisher Wiley & Sons.

For more information contact Kate Guillemette, Product Development Editor, at kguillemette@computer.org.

 **CS Press**
www.computer.org/cspress

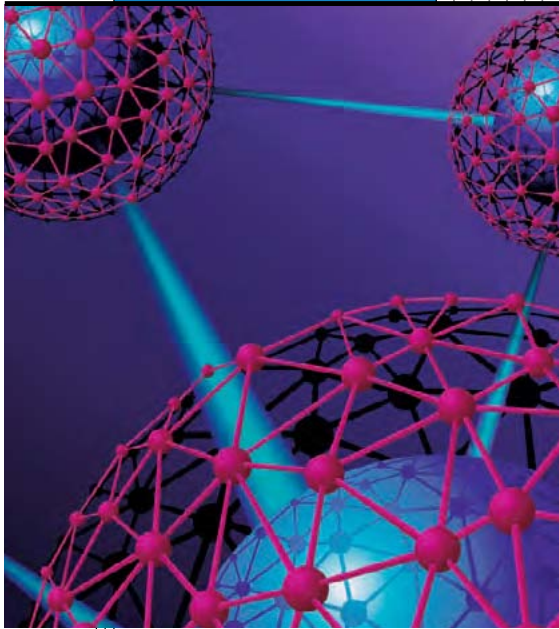
 **WILEY**
Publishers Since 1807

Evolving Software Architecture Descriptions of Critical Systems

by Tom Mens, Jeff Magee, and Bernhard Rumpe

Computer, vol. 43, no. 5, May 2010, pp. 42–48

DOI bookmark: <http://doi.ieeecomputersociety.org/10.1109/MC.2010.136>



EVOLVING SOFTWARE ARCHITECTURE DESCRIPTIONS OF CRITICAL SYSTEMS

Tom Mens, *Université de Mons, Belgium*

Jeff Magee, *Imperial College London, UK*

Bernhard Rumpe, *RWTH Aachen University, Germany*

To manage the complexity of developing, maintaining, and evolving a critical software-intensive system, its architecture description must be accurately and traceably linked to its implementation.

Software-intensive systems, whether real-time embedded systems or information-processing systems, present critical concerns for stakeholders. A system may be mission-critical for a company, in that it could lose its competitive advantage or even be unable to survive if the system doesn't function properly. A system may be resource-critical in terms of time, personnel, hardware, or any other crucial resource on which it might rely; unavailability or malfunction of these resources could cause the system to fail. A system may be critical in a more traditional sense—having specific nonfunctional characteristics that must be satisfied at all times. For example, financial systems are security-critical, whereas nuclear power plants, medical applications, and public transportation are safety-critical, as human lives might be at stake.

Software architectures provide a sound basis for explicitly documenting these concerns. IEEE standard 1471-2000, which has also become ISO/IEC 42010:2007, recommends

providing architectural descriptions of software-intensive systems to cope with their increasing complexity and to mitigate the risks incurred in constructing and evolving these systems. According to this standard,¹ as Figure 1 shows, a system fulfills a particular mission in the environment it inhabits and has one or more stakeholders that have concerns relative to the system and its mission. Concerns are defined as “those interests that pertain to the system's development, its operation, or any other aspects that are critical or otherwise important to one or more stakeholders.” Runtime concerns include performance, reliability, security, and distribution; development concerns focus on maintenance—in particular, evolvability.

The software architecture deals with multiple views of a system including both its functional and nonfunctional aspects. A structural view looks at the system as a set of components that interact via connectors. Complexity is mastered by means of hierarchical decomposition; a component can be composed from subcomponents with the hierarchy's leaf components representing coded functionality. As the “Architecture Description Languages” sidebar describes, the research community has proposed numerous ADLs, some of which have found their way into commercial practice.

An explicit architecture description is important but not sufficient to manage the complexity of developing, maintaining, and evolving a critical software-intensive

system. The description must also be accurately and traceably linked to the software's implementation, so that any change to the architecture is reflected directly in the implementation, and vice versa. Otherwise, the architecture description will become rapidly obsolete as the software evolves to accommodate changes. The architecture description must thus be an integral part of the software-intensive system and its documentation.

WHY EVOLVE ARCHITECTURE DESCRIPTIONS?

Any software-intensive system is constantly subject to software changes, usually driven by external stimuli from the system environment over which the developers have little or no control. These stimuli may be as diverse and unforeseeable as technological changes, enhanced user

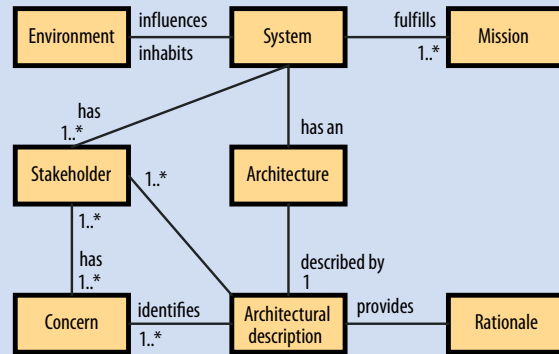


Figure 1. Fragment of IEEE Std. 1471 conceptual model of architectural description. A software-intensive system fulfills a particular mission in the environment it inhabits and has one or more stakeholders that have concerns relative to the system and its mission.

➔ ARCHITECTURE DESCRIPTION LANGUAGES

ADLs have emerged as formal languages to define and document the software architecture of systems.¹⁻⁴ They facilitate communication between software architects and other stakeholders and make it possible to express, verify, and impose properties upon the software that will implement the architecture. In contrast to programming languages, ADLs are usually declarative and describe a system's architecture as a set of components, connectors, and configurations of these elements.

Researchers have developed numerous ADLs such as AADL (Architecture Analysis and Design Language), Acme, COSA (Component Object-based Software Architecture), Darwin, Rapide, and Wright. Appropriate architecture-centric software development tools have also been developed, including ArchStudio, AcmeStudio, and SafArch Studio.

Koala⁵ is one of the few ADLs to have found application in commercial practice. Philips uses it to define the software architecture for consumer electronic products. Koala is model-driven in that it directly uses the architectural description to construct the software loaded into products.

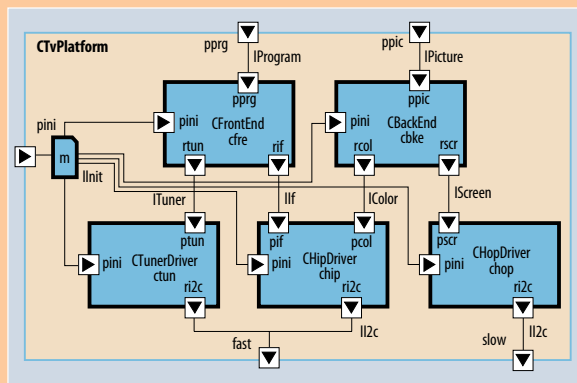


Figure A. Architectural description of the software for a TV set using Koala. The components can be configurations of more primitive components or they can be base-level components with their implementations defined in C.

Figure A⁵ shows an example of the architectural description of the software for a TV set using Koala. The components shown in the figure can be configurations of more primitive components or they can be base-level components with their implementations defined in C. This ability to describe systems as hierarchical compositions of components is the key to managing complexity and is a feature of practically all ADLs.

In the figure, the boxes with arrows represent interfaces defined by sets of function calls. If the arrow points into a component, then the component provides or implements that interface; if it points out of the box, then the component requires access to the interface. The lines or connectors represent connections between required and provided interfaces and represent runtime function call paths. Connectors in other ADLs represent more general connector semantics that can encompass streams, events, and message-passing protocols.

Koala restricts itself to a structural description of software architecture. However, much of the power of ADLs and their importance to critical systems arises from the ability to associate behavioral, functional, and nonfunctional properties with components and reason about the preservation of overall system properties.

With the advent of Unified Modeling Language v. 2.x, more modern ADL proposals are essentially profiles that extend UML 2.x by means of stereotypes to extend the existing UML 2.x structural elements with additional properties and constraints.

References

1. R.N. Taylor, N. Medvidovic, and E.M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, Wiley, 2009.
2. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
3. N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.*, Jan. 2000, pp. 70-93.
4. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
5. R. van Ommering et al., "The Koala Component Model for Consumer Electronics Software," *Computer*, Mar. 2000, pp. 78-85.

organizational structures or business processes, new legislation, or changes in resources.

To cope with any of these issues, all software artifacts produced and used by the software-intensive system must evolve. Depending on the software artifacts' type and granularity, the impact and rate of change may differ. Source-code artifacts need to be changed frequently—for example, to fix bugs—but often have a local impact only. Changes to the architecture occur less frequently but have a global impact.

Evolving a software architecture by modifying its description to accommodate change requests faces numerous research challenges. In particular, the evolution of an architectural description should typically preserve its purpose and criticality concerns. There are two ways to verify that such properties are preserved: by analyzing and verifying the resulting architectural description after the changes, or by analyzing the initial architectural description together with the “delta” or “increment” applied to it to make the changes.



The evolution of an architectural description should typically preserve its purpose and criticality concerns.

Current ADLs provide little support for architectural evolution, leaving it to processes, tools, and techniques outside the architecture description's concern.² Nevertheless, researchers agree that evolving the architecture description is beneficial, particularly in the case of critical systems, and in recent years have made promising gains.

MODEL-TRANSFORMATION-BASED EVOLUTION

The model-driven-engineering community uses models as artifacts to describe well-defined software aspects at a higher abstraction level than source code. *Model transformation* is a well-established technique to modify and evolve models.³ Researchers have developed various model-transformation languages, some of which—such as ATL (ATLAS Transformation Language)—are seeing widespread industry adoption. Others are part of a standardization process, such as QVT (Query/View/Transformation), the de facto standard proposed by the Object Management Group to accompany UML (Unified Modeling Language). Because an architectural description can be seen as a software model, it makes sense to apply model-transformation approaches to architectural evolution.

Developers are applying the proven program-transformation technique of refactoring to models and specifications as well. Due to their semantic richness,

models are often easier to evolve than programs. For almost any modeling language, various techniques exist to systematically modify the models to achieve certain effects. For example, composite structure diagrams can be transformed and refined⁴ in a semantic-preserving way.

Many researchers have studied the formal foundations of model transformation. One well-known formalism used for this purpose is *graph transformation*, which enables reasoning about the formal properties of model transformations—in particular, how an architecture evolves. For example, this approach can be used to verify whether a given architectural transformation preserves certain structural, behavioral, or other properties. This is particularly useful in the context of architectural restructuring, which aims to improve the structure of an architectural description while improving its behavioral properties.

Using model transformation, and especially graph transformation, to express and formalize the evolution of architectural descriptions isn't new. Daniel Le Métayer⁵ proposed such an approach more than a decade ago. More recently, Michael Wermelinger and José Luiz Fiadeiro⁶ used graph transformation theory as a formal foundation for software architecture reconfiguration. Even more recently, Lars Grunske⁷ formalized architectural refactorings as graph transformations that can be applied automatically. In a similar vein, Dalila Tamzalit and one of the authors⁸ used graph transformations to express architectural evolution patterns as a means to introduce architectural styles as well as to verify whether a given architectural evolution preserves the constraints imposed by an architectural style. Automated support for this approach is currently under development using the COSA ADL and associated tools.

Another interesting approach to transformation-based architectural evolution, though not directly relying on graph transformation, is work by Olivier Barais and colleagues.² Their TranSAT framework supports architectural evolution based on ideas borrowed from aspect-oriented software development. The idea is to encapsulate new architectural concerns as architectural aspects and to use an architectural-transformation language to weave these aspects into the existing architecture description. This approach makes it possible to analyze transformations statically and incrementally to verify whether the resulting architecture description is structurally consistent—this saves considerable time and effort compared to doing a complete analysis of the resulting architecture description. Examples of such architectural restructuring include the transformation of a monolithic architecture into a distributed client-server architecture or into a three-tiered architecture that clearly separates the user interface, business logic, and data layer.

ARCHITECTURAL COEVOLUTION

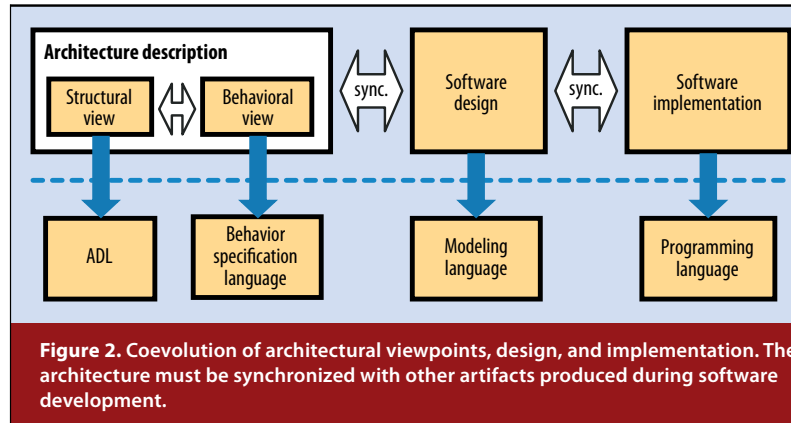
While in many disciplines architectural descriptions are primarily concerned with structure, architectural descriptions of software serve as structural containers in which the complex behavior resides. From the end-user viewpoint, achieving correct and reliable behavior and functionality is the ultimate goal of a critical software-intensive system. The internal structure is only relevant to the software architects and developers who use it to master the software complexity. To reconcile both types of stakeholders, we need different views to represent the structural and behavioral descriptions of architecture.

Behavioral descriptions are often modeled in a precise formal form. Various modeling languages such as state-machine diagrams, sequence and activity diagrams, Petri nets, and temporal or other forms of logic are used to describe a system's behavioral aspects. All these behavioral languages either incorporate their own structural description or can be combined with a separate one expressed using some ADL or modeling language.

Evolving architectural descriptions inevitably requires the *coevolution* of different viewpoints: the structural viewpoint, the behavioral viewpoint, and often many other viewpoints as well. In addition, as Figure 2 shows, the architecture must be synchronized with other artifacts produced during software development such as system requirements, documentation, and, of course, implementation.

While most modeling languages have transformation techniques to evolve models in small, understandable steps, keeping models synchronized remains a challenge. Tool chains currently translate all models into a logic language and feed that into a verifier, but this clumsy technique fails to capture the modeling language's semantic richness and structure, and a modified model often can't be translated back into the original model.

Understanding how to transform structural descriptions and accompanying behavioral models in a synchronized, consistent way is critical to software development. Even more important is the coevolution of analysis or certification arguments, which can retain already validated properties if not affected directly. Proof-replay techniques for verifiers have had some success in this regard. However, researchers don't yet grasp how heterogeneous modeling languages semantically fit together or how to consistently coevolve them. This is especially true for structural ADLs and behavioral



models. It's even unclear how state-, activity-, and flow-based models of the same architecture complement one another.

PRESERVING CRITICAL BEHAVIORAL PROPERTIES

It's essential to ensure that any evolutionary software adaptation retains desired properties that have been modeled, validated with stakeholders, or even formally proven correct versus requirements and implementation. This is even more important for critical systems, in which errors are often introduced during badly managed evolutionary steps. Making large architectural changes in one step is especially problematic. After such a "big bang," considerable validation and modification must occur to adapt behavioral models as well as any implementation. In contrast, a stepwise approach to evolution lets developers manage change more effectively through small, incremental transformations.


Transformations that refine or preserve behavior while adapting the architectural description to new requirements or technical needs are relatively complex, even in small evolutionary steps. Tools are therefore necessary to assist such transformations. Unfortunately, none of today's tools adequately preserve syntactical correctness and semantics. Further, researchers have mainly applied them to isolated modeling viewpoints and not to loosely coupled heterogeneous views, which are needed to describe an architecture's structure and behavior.

Transformation-based evolution of behavioral models is much harder to achieve than evolution of purely structural models. Tools usually carry out structural transformations rather efficiently. When behavior is involved, however, undecidability problems pop up such as semantic equivalence of logical preconditions. A simple solution to these problems would be to review them by hand; the most complex would be to feed them into an interactive verifier and enforce their formal correctness proof. This is why evolution techniques for behavior in

architecture descriptions will first arise only in certain kinds of critical systems.

A less expensive alternative is to use automated tests and invariants to iteratively check whether each evolution step is carried out correctly. However, this raises another problem: When evolving software architecture based on architectural descriptions, how do you keep the architecture consistent with the implementation?

One way to keep architectural artifacts consistent during evolution is to trace information-flow dependencies through them. Horizontal tracing aims to ensure consistency between architectural descriptions at the same stage of development, while vertical tracing aims



It's impossible, whichever development process is adopted, to foresee all possible future requirements for evolving a system.

to maintain consistency between the stages of development—for example, by aligning artifacts with code. Informal tracing is difficult because dependencies are easy to forget. Formal tracing techniques exist—for example, to formally check source-code annotations.⁹ Explicitly adding evolution operators to the language helps to alleviate this problem, as the original information is still available and no trace is needed to recover dependencies. The optimal solution would be to generate parts of the code in such a form that it can be regenerated after each evolutionary step; automated tests could then regressively test system behavior.

ARCHITECTURAL CHANGE AS A FIRST-CLASS CONSTRUCT

Current ADLs such as Koala don't directly address evolution, regarding it as extrinsic to architectural descriptions. The alternative is to provide first-class structural constructs to express and capture architectural change during both initial development and subsequent evolution. This necessitates dealing with unplanned modification, for it's impossible, whichever development process is adopted, to foresee all possible future requirements for evolving a system. While this approach may initially seem unusual, some programming languages already contain explicit constructs for system evolution. For example, subclassing could be interpreted as a form of evolution of classes where the "old" class taken from the library isn't evolved but adapted through the subclass only. However, subclassing permits only conservative extension—adding elements to but not removing them from a class.

Designer dilemma

Unplanned evolutionary change introduces a dilemma when designing built-in ADL language constructs to support change and extension. On one hand, constructs that always result in structurally well-formed and type-correct systems would inevitably permit only a subset of all possible valid system changes. On the other, constructs that result in invalid systems could only be permissible in an environment that comprehensively detects structural problems and type errors, especially with critical systems. There is thus a need to combine the freedom to perform incorrect changes with the ability to detect these errors to achieve sufficient expressiveness for unplanned changes. This comprehensive approach can accommodate destructive change—deleting elements from an architecture description—in addition to constructive change—adding elements to an architecture description.

When defining architectural changes as a first-class construct in an ADL, software architects should consider the different requirements of organizations responsible for system development, deployment, and modification. Consider, for example, a common scenario in the domain of enterprise resource planning software. A development organization produces a software framework product used by other organizations to build applications. To meet their local development requirements, these organizations may need to customize (modify and extend) the framework to support their applications. The original framework will evolve over time, so the organizations that use it must apply their local changes to the framework before using the evolved framework for their applications. In addition, a third party might wish to use applications from more than one framework customizer and thus needs to merge changes from both these organizations and the original framework provider.

Regarding an architecture description only as design documentation leads to the coevolution problem shown in Figure 2: keeping this documentation in synch with the software implementation as the system evolves. A model-driven-engineering approach ensures that an architecture definition isn't just a documentation artifact but a precise model for constructing both initial implementations and extensions to these implementations.

Example: Resemblance and replacement

Figure 3 illustrates two techniques, *resemblance* and *replacement*, that can be used to extend UML 2.x to permit the intrinsic definition of architectural evolution.¹⁰

Resemblance defines a new component as the difference in structure from one or more existing components. It's the delta—the set of additions, deletions, and replacements—of the components' elements applied to arrive at the new definition. Component elements include

- *parts*—instances of subcomponents,
- *ports*—instances of interfaces,
- *connectors*—bindings between ports, and
- *attributes*—component parameters.

Resemblance can also be applied to interfaces, in which case the modified elements are *operations*. If a resemblance delta consists only of additions, then when applied to an interface, it defines a proper subtype and thus can safely replace the original component.

Figure 3a depicts the architecture description of a simple database server that has two internal component parts: Database and FrontEnd. Figure 3b shows an evolution of this simple server that has been extended using resemblance to add managed access to the data stored in the server. ManagedServer resembles Server, and the text note defines the delta that results from editing Server to arrive at ManagedServer.

Resemblance's many-to-one relation permits the merging of multiple component definitions that may have arisen due to, for example, distributed development. Applying a sufficiently radical delta to a component may result in a new definition that bears little or no resemblance to the component definitions from which it's derived. Tracing evolutionary origins remains very important in many project contexts, as both engineering and nature provide many examples of systems that have dramatically evolved from their original form.

Replacement globally substitutes the definition of one component for another while preserving the original definition's identity, thereby maintaining any relations that a larger system has with this component. Combined with resemblance, replacement permits the incremental evolution of a component definition without having to change the composite definitions that use this component. Re-

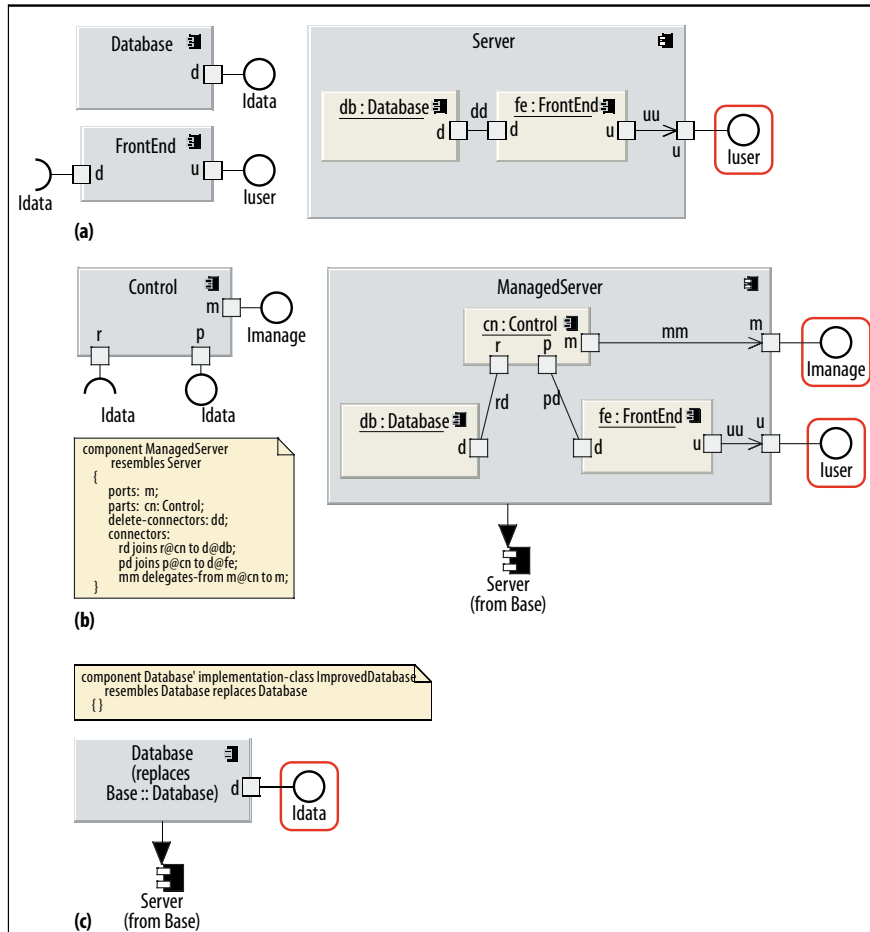


Figure 3. Evolving a software architecture description using Evolve, a UML 2.x evolution tool developed by Andrew McVeigh. (a) Architecture description of a simple database server. (b) Resemblance: architecture description of managed database server. (c) Replacement: replacing the Database component.

placement is the key to managing change in composite hierarchical definitions because it enables substitution of definitions at one level of the hierarchy without necessarily affecting higher layers. For example, Figure 3c shows an improved implementation of the Database component that replaces the original Database when applied to the simple database server system (Figure 3a) or the managed server system (Figure 3b).

Resemblance allows elements to be deleted in forming a new definition from existing ones, but it isn't destructive editing in the traditional sense. Using resemblance to replace a definition in a base model with a new definition in an extension model doesn't remove the old definition; instead, it records the deletion in a delta. This approach enables history tracing, the use of base models instead of derivatives, and the resolution of conflicts when independently evolved extensions are subsequently merged.

Incremental change is integral to both the initial development and subsequent evolution of software-intensive critical systems. Making evolution intrinsic to architecture description is a principled and manageable way to deal with unplanned change. This intrinsic definition facilitates decentralized evolution of software by multiple independent developers. Unplanned extensions can be deployed to end users with the same facility that plug-in extensions are currently added to systems with planned extension points. **E**

Acknowledgments

Tom Mens is supported by ARC project AUWB-08/12-UMH19, "Model-Driven Software Evolution," funded by the Ministère de la Communauté française—Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, and by the project TIC, cofunded by the European Regional Development Fund (ERDF) and the Walloon Region (Belgium).

References

1. IEEE Std. 1471-2000 and ISO/IEC 42010:2007, *Recommended Practice for Architectural Description of Software-Intensive Systems*, 2007.
2. O. Barais et al., "Software Architecture Evolution," *Software Evolution*, T. Mens and S. Demeyer, eds., Springer, 2008, pp. 233-262.
3. S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no. 5, 2003, pp. 42-45.
4. J. Philipps and B. Rumpe, "Refinement of Pipe-and-Filter Architectures," *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM 99)*, LNCS 1708, Springer, 1999, pp. 96-115.
5. D. Le Métayer, "Describing Software Architecture Styles Using Graph Grammars," *IEEE Trans. Software Eng.*, vol. 24, no. 7, 1998, pp. 521-533.
6. M. Wermelinger and J.L. Fiadeiro, "A Graph Transformation Approach to Software Architecture Reconfiguration," *Science of Computer Programming*, vol. 44, no. 2, 2002, pp. 133-155.
7. L. Grunske, "Formalizing Architectural Refactorings as Graph Transformation Systems," *Proc. 6th Int'l Conf. Software Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing and 1st ACIS Int'l Workshop Self-Assembling Wireless Networks (SNPD/SAWN 05)*, IEEE CS Press, 2005, pp. 324-329.
8. D. Tamzalit and T. Mens, "Guiding Architectural Restructuring through Architectural Styles," *Proc. 17th Ann. IEEE Int'l Conf. and Workshop Eng. of Computer-Based Systems (ECBS 10)*, IEEE Press, 2010, pp. 69-78.
9. H. Krahne and B. Rumpe, "Towards Enabling Architectural Refactorings through Source Code Annotations," *Proc. der Modellierung 2006*, Gesellschaft für Informatik, 2006, pp. 203-212.
10. A. McVeigh, J. Kramer, and J. Magee, "Using Resemblance to Support Component Reuse and Evolution," *Proc. 2006 Conf. Specification and Verification of Component-Based Systems (SAVCBS 06)*, ACM Press, 2006, pp. 49-56.

Tom Mens is a professor and directs the Software Engineering Lab at the Institut d'Informatique, Faculty of Sciences, Université de Mons, Belgium. His research interests are in formal foundations and automated tool support for software evolution. Mens received a PhD in sciences from Vrije Universiteit Brussel, Belgium. He is a member of IEEE, the IEEE Computer Society, the ACM, the European Research Consortium for Informatics and Mathematics (ERCIM), and the European Association of Software Science and Technology (EASST). Contact him at tom.mens@umons.ac.be.

Jeff Magee is a professor, and heads the Department of Computing, at Imperial College London, UK. His research interests include software architecture, distributed systems, and mobile computing. Magee received a PhD in computer science from Imperial College London. He is a Chartered Fellow of the British Computer Society. Contact him at j.magee@imperial.ac.uk.

Bernhard Rumpe is a professor of software engineering in the Department of Computer Science at RWTH Aachen University, Germany. His research interests include modeling, software architecture, and evolution. Rumpe received an Habilitation in computer science from Munich University of Technology (TUM). He is a member of the IEEE Computer Society, the ACM, and Gesellschaft für Informatik (GI). Contact him at rumpe@se-rwth.de.

cn Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.

2 Free Sample Issues!

A \$26 value



The magazine of computational tools and methods for 21st century science.

<http://cise.aip.org>
www.computer.org/cise

Send an e-mail to jbebee@aip.org to receive the two most recent issues of CISE. (Please include your mailing address.)

MEMBERS
\$47/year
for print & online



Recent Peer-Reviewed Topics:

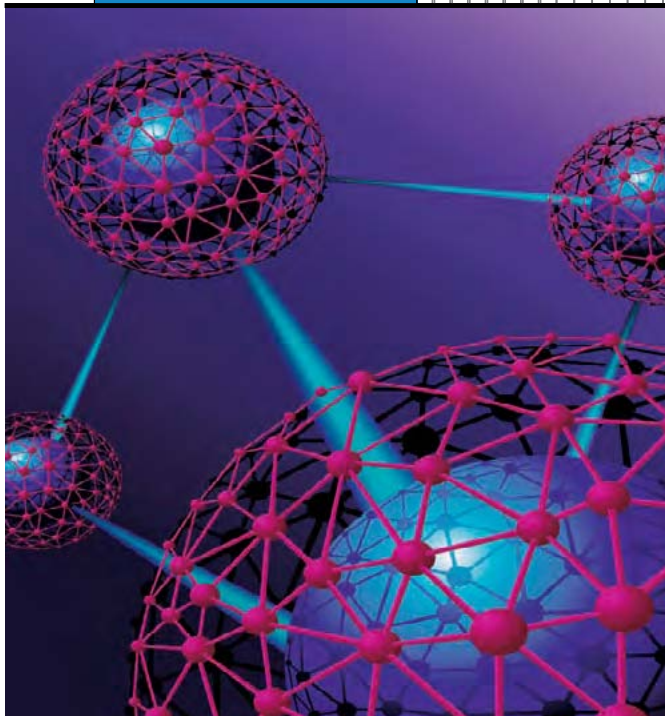
Cloud Computing
Computational Astrophysics
Computational Nanoscience
Computational Engineering
Geographical Information Systems
New Directions
Petascale Computing
Reproducible Research
Software Engineering

Evolution in Relation to Risk and Trust Management

by Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen

Computer, vol. 43, no. 5, May 2010, pp. 49–55

DOI bookmark: <http://doi.ieeecomputersociety.org/10.1109/MC.2010.134>



EVOLUTION IN RELATION TO RISK AND TRUST MANAGEMENT

Mass Soldal Lund and Bjørnar Solhaug, *SINTEF ICT*
Ketil Stølen, *SINTEF ICT and University of Oslo*

A methodology within risk and trust management in general, and risk and trust assessment in particular, isn't well equipped to address trust issues in evolution.

When improving an existing methodology to account for evolution, we must realize that methodological needs are strongly situation dependent. We therefore distinguish among three main assessment scenarios, each giving a particular perspective in relation to risk and trust assessment: *maintenance*, *before-after*, and *continuous-evolution*. For each perspective, we identify its main methodological challenges.

A risk picture typically focuses on a particular system configuration at a particular point in time and is thus valid only under the assumptions made when it was established. However, the system and its environment, as well as our knowledge, tend to evolve over time. State-of-the-art methodologies within risk management in general, and risk assessment in particular, aren't well-equipped to address evolution. A risk management standard such as ISO 31000^{1,2} prescribes change detection and identifi-

cation for emerging risks, but provides no guidelines. An important risk assessment methodology like OCTAVE³ recommends reviewing risks and critical assets, but responds with silence when addressing how risk assessment results should be updated. Moreover, most academic studies have focused on either maintenance^{4,5} or variants of reassessment.^{6,7}

Matt Blaze⁸ coined the term *trust management* in 1996, calling it a systematic approach to managing security policies, credentials, and trust relationships regarding authorization and delegation of security-critical decisions. Trust management has since been the subject of increased attention and today provides for a diversity of approaches. We view trust management as risk management with a special focus on understanding the impact that subjective trust relations within and between a target and its environment have on the target's factual risks. A methodology for trust management suffers from the same weaknesses we've identified for risk management and, further, brings in additional challenges due to trust's complexity and dynamic nature.

RISK MANAGEMENT

The recently published risk management standard ISO 31000^{1,2} defines risk management as coordinated activities

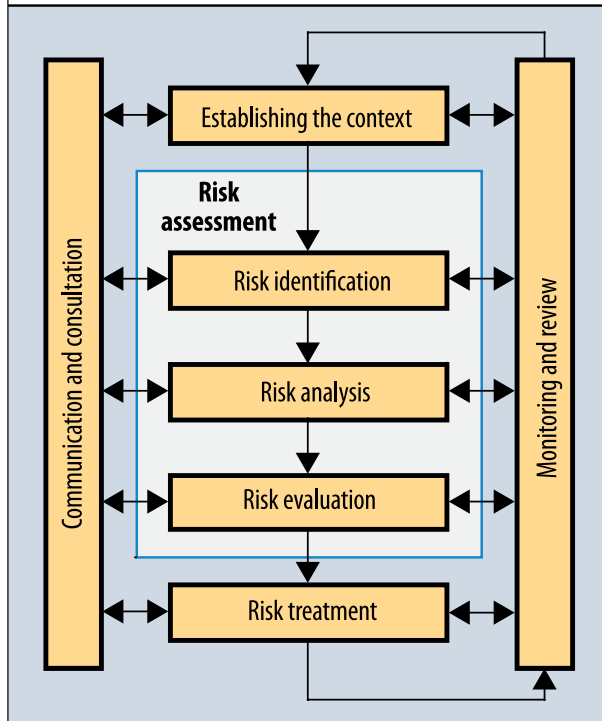


Figure 1. Risk management process. ISO 31000 defines risk management as coordinated activities to direct and control an organization's risk, defined as a combination of an event's consequences and their associated likelihood.

to direct and control an organization's risk, defined as a combination of an event's consequences and its associated likelihood. The risk management process is defined as the systematic application of management policies, procedures, and practices to the activities of communicating, consulting, establishing context, and identifying, analyzing, evaluating, treating, monitoring, and reviewing risk. Figure 1, from *ISO 31000, Risk Management: Principles and Guidelines*,¹ shows the risk management process's seven subprocesses.

Seven subprocesses define risk management as a series of coordinated activities, as follows:

- *Establishing the context* defines the external and internal parameters to be accounted for when managing risk, and sets the scope and risk criteria for the risk management policy.
- *Risk identification* finds, recognizes, and describes risks.
- *Risk analysis* comprehends the nature of risk and determines its level.
- *Risk evaluation* compares the results of risk analysis with risk criteria to determine whether the risk and its magnitude are acceptable or tolerable.

- *Risk treatment* is the process of modifying risk.
- *Communication and consultation* are the continual and iterative processes an organization conducts to provide, share, or obtain information and to engage in dialogue with stakeholders about risk management.
- *Monitoring* involves continually checking, supervising, and critically observing risk status to identify changes from the performance level required or expected, whereas *review* focuses on the activity undertaken to determine the suitability, adequacy, and effectiveness of the subject matter necessary to achieve established objectives.

The *monitor and review* subprocess supposedly detects “changes in the external and internal context, including changes to risk criteria and the risk itself, which can require revision of risk treatments and priorities.”¹ Hence, ISO 31000 covers evolution, but we must still address evolution in the more technical risk management activities, particularly the three subprocesses that Figure 1 refers to as *risk assessment*.

EVOLUTION IN RELATION TO RISK ASSESSMENT

A risk assessment as traditionally performed focuses on a particular target configuration at a particular point in time, and is thus valid only under the assumptions made when conducting the assessment. Because systems and environments change, we need more powerful risk assessment methodologies that can address changing and evolving targets.

How we should handle change and evolution in relation to risk assessment depends greatly on the context and kind of changes we face:

- Do the changes result from maintenance or from bigger, planned changes?
- Do the changes comprise a transition from one stable target state to another, or do they reflect the continuous evolution of a target designed to change over time?
- Do the changes occur in the target or in the target's environment?

The answers to such questions, as well as the risk assessment's practical setting, decide the methodological needs.

Maintenance perspective

We can describe the scenario corresponding to the maintenance perspective in the following example: risk assessors conducted an assessment three years ago and are now requested by the same client to reassess and update the risk picture to reflect changes to the target or environment, thereby restoring the assessment's validity.

The changes we address from the maintenance perspective are those that accumulate more or less unnoticed over time. Such changes can be bug fixes and security patches, an increase in network traffic, or an increase in the number of attacks. In this case, the risk picture remains more or less the same, but risk values might have changed such that previously acceptable risks could now be unacceptable, or vice versa. The objective then becomes maintaining the previous risk assessment's documentation by conducting an update.

Figure 2 shows the principle by which risk assessors conduct such a reassessment from the maintenance perspective. Assuming that we have descriptions of the old target and the updated target available, including environment descriptions, we start by identifying the changes that have occurred. We then use the relevant changes as input to the risk reassessment when deriving the current risk picture.

From a methodological viewpoint, the main challenge involves reusing the old risk assessment and avoiding a restart from scratch. This demands identifying the updates made to the target, updating the target description accordingly, and identifying which risks—and hence which parts of the risk picture—the updates affect. Finally, we update the risk picture without making changes to the unaffected parts.

Before-after perspective

The motivating scenario for the before-after perspective is risk assessors that are asked to predict the effect that implementing changes in the target has on the risk picture.

The changes we address from the before-after perspective are planned and anticipated, but could still be radical. Such changes can, for example, involve rolling out a new system or making major organizational changes such as implementing a merger agreement between two companies. We thus must understand the current risk picture, the risks that might arise from the very process of change, and the future risk picture.

Figure 3 shows the principle by which we conduct a risk assessment from the before-after perspective. Assuming we have descriptions of the current target and the change process to bring it from the current to the future state, we can devise a coherent risk picture for the future target and the change process.

From a methodological viewpoint, the main challenges involve obtaining and presenting a risk picture that unambiguously describes the current and future risks and the impact of the change process itself. This requires an approach for presenting a target description that unambiguously characterizes the target both “as is” and “to be,” specifying the process of change in sufficient detail, identifying current and future risks without doing double work, identifying risks due to the change process.

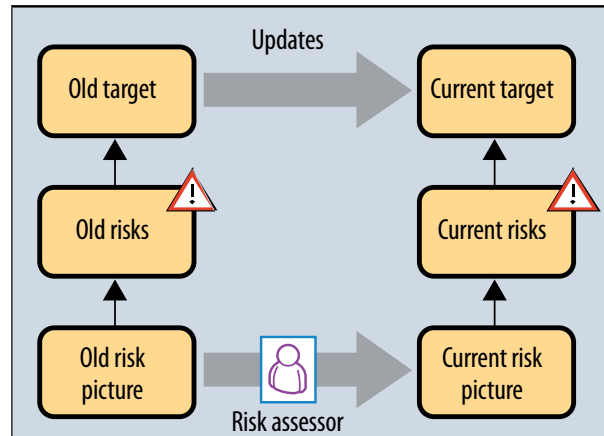


Figure 2. Maintenance perspective. Assuming we have descriptions of the old target and updated target available, including environment descriptions, we start by identifying the changes that have occurred in between, and then use the relevant changes as input to the risk assessment when deriving the current risk picture.

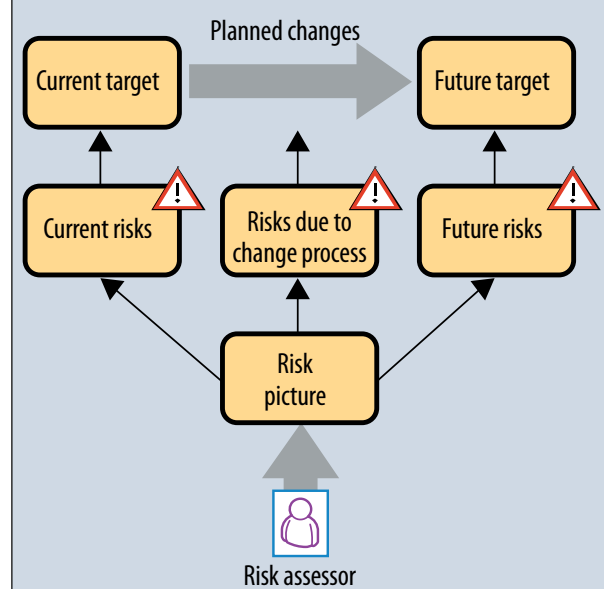


Figure 3. The before-after perspective. Assuming that descriptions of the current target and the change process bring the target from the current to the future state, we can devise a coherent risk picture for the future target and the change process.

Continuous-evolution perspective

The continuous-evolution perspective applies in the scenario that risk assessors are requested to predict future evolution of risk. It mandates that risk assessors conduct an assessment that establishes a dynamic risk picture reflecting the target's expected evolution. The changes we

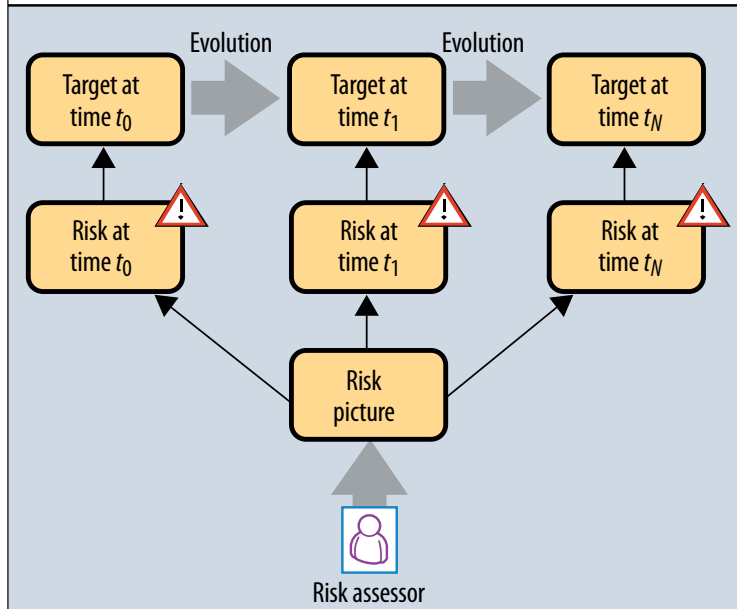


Figure 4. Continuous-evolution perspective. Given a description of the target as a function of time that we can derive at any point, we use this information to inform the risk assessment. Knowing how the target and its environment will evolve, we can create a risk picture as a function of time that describes how risks evolve.

address from the continuous-evolution perspective are predictable and gradual evolutions, described as functions of time. These predictions can be based on well-founded forecasts or planned developments. Examples include the slow increase in the number of components working in parallel, or gradually including more sites in a system. Examples of well-founded forecasts can include the expected steady increase of end users, adversary attacks, and annual turnover.

Figure 4 shows the principle by which we conduct a risk assessment from the continuous-evolution perspective. Assuming that we have a description of the target as a function of time, such that we can derive the target at any point, we use this as input to the risk assessment. Knowing how the target and its environment evolve, we seek to craft a risk picture as a function of time that shows how risks evolve.

From a methodological viewpoint, the main challenges are how to identify evolving risks and present them in a dynamic risk picture. Obtaining this information requires that we generalize the target description such that it characterizes the evolution of the target and its environment, identify and generalize the risks affected by evolution, characterize the evolution of risks in the dynamic risk picture, and relate the evolution of risks to the target's evolution as captured by the target description.

TRUST MANAGEMENT VS. RISK MANAGEMENT

Researchers agree that trustworthiness is a more general issue than risk-related factors such as dependability, security, and safety. For example, although the underlying system could be completely dependable in the traditional sense, it might not be trustworthy unless a suitable legal framework exists on which the trustor can rely should problems arise. Trust is nevertheless inherently related to risk, and an important part of managing trust is understanding the risks involved in trust-based interaction.

Following the example of Diego Gambetta⁹ and Audun Jøsang and colleagues,¹⁰ we define trust as the subjective probability by which the trustor expects that another entity—the trustee—performs a given action on which the actor's welfare depends. By this definition, trust is a belief the trustor holds about the trustee with respect to a particular action as a probability ranging from 0 (complete distrust) to 1 (complete trust). The trustor's welfare refers to its assets. If the trustee performs as expected, it might have a positive effect on the trustor's welfare; otherwise, it might have a negative effect.

The positive and negative outcomes correspond to opportunity and risk, respectively.^{11,12} Issues of trust arise when deception or betrayal are possible, creating an inevitable relation between trust and risk. Likewise, trust always relates to opportunity, which is risk's counterpart. In a trust-based transaction, the trustor might be willing to accept the risk considering the opportunities involved.

We can calculate the risk level as a function R from the consequence (loss) l of a harmful event and the probability p of its occurrence. We define the dual notion of opportunity as the combination of the gain and likelihood of a beneficial event, and give the level of opportunity as a function O from the gain g of the beneficial event and the probability p of its occurrence.

Assume that the trustor has trust level p in the trustee performing an action with gain g for the trustor and that deception has loss l . The trustor must then weigh the opportunity $O(g, p)$ and risk $R(l, 1 - p)$ against each other when deciding whether to engage in the trust-based interaction. For example, assume a situation in which the trustor considers lending \$80 to the trustee, with the promise of being repaid the amount with 50 percent interest, a gain of \$40. The trust level is 0.9. Using multiplication as the risk and opportunity functions, the opportunity level is $0.9 \times 40 = 36$, and the risk level is $0.1 \times 80 = 8$. Because the

opportunity outweighs the risk, the trustor should accept the transaction.

Trust is just a belief held by the trustor, so the estimated trust level might be wrong and so too might the subjectively estimated levels of risk and opportunity. Trust is important precisely for decisions that must or should be made, even when confronting a lack of evidence about the trustee's future behavior. To precisely assess and evaluate trust-based decisions, however, the trustor's belief and the basis for it must be considered.

We say that trust is well-founded if the trustor's assessment equals the trustee's trustworthiness—that is, the objective and factual probability by which the trustee performs a given action on which the trustor's welfare depends. Only in the case of well-founded trust can the trustor correctly estimate the involved risks and opportunities. If trust is ill-founded, there's a chance of misplacing it. If the trust level is higher than the trustworthiness, the transaction might be at greater risk than the trustor believes. On the other hand, if the trust level is lower, distrust is misplaced, and the actual risk is lower than believed. To continue the example, assume the trustor's trustworthiness with respect to the transaction in question is only 0.65. The factual opportunity level is then $0.65 \times 40 = 26$, and the factual risk level is $0.35 \times 80 = 28$, making the risk higher than the opportunity.

Three focal points of trust management

In today's information society, traditionally face-to-face or human-to-human interactions are increasingly conducted remotely over the Internet. Moreover, computerized agents communicate and negotiate based on policies resembling those of humans. Because trust often is a precondition for such interactions to take place, trust must be managed. The adequate or appropriate approach, however, depends on the particular viewpoint and setting. Specifically, we must distinguish among three different focal points that might require less systematic management—namely, trust management from the focal point of the trustor, the trustee, and risk management.

From the trustor's focal point, there's a need to assess the trustworthiness of other entities to make trust-based decisions. From the trustee's focal point, there's a need to increase and correctly represent the trustee's trustworthiness as well as its systems and services.¹⁰ The third focal point, trust management in the setting of risk management, is an important concern and involves understanding the impact of trust on the target's factual risk picture. The target then includes actors that base some of their decisions on trust, wherein the trust relations might be both within the target and between the target and its environment. These actors could be human, but they might also be organizations, businesses, or computerized entities behaving on behalf of other actors.

When conducting trust management from the focal point of risk management, we seek to direct and control an organization with regard to the risk and opportunity that stems from trust relations. To appropriately address and assess trust in this setting, we must generalize the risk management process depicted in Figure 1 by making the corresponding trust assessment steps accompany the identified risk assessment steps:

- *Identification of trust relations* focuses on existing and potential trust relations that might serve as a basis for trust-based decisions of actors within the target.
- *Trust analysis* estimates the trustee's trustworthiness in each such relation and estimates the potential for gain and loss for each potential trust-based decision. The trust analysis also includes an evaluation of the extent to which trust is well-founded.



Common for any trust management in the risk management setting is the dynamic and evolving nature of trust.

- *Trust evaluation* determines the risk and opportunity levels associated with the trust relations and thereby identifies favorable and unfavorable trust-based decisions.

The final risk management step should also be generalized to include strategies that ensure the actor makes only beneficial trust-based decisions in which opportunity outweighs risk. Such a strategy can, for example, be specified and enforced as a trust policy. A strategy to ensure well-founded trust should also be identified in case the trust analysis reveals significant discrepancy between trust and trustworthiness.

EVOLUTION IN RELATION TO TRUST MANAGEMENT

We can classify evolution in relation to trust management into the same three perspectives as evolution in relation to risk management. It is, however, more challenging because we must consider that trust relations are highly dynamic and can evolve as any other feature of the target; moreover, we must contemplate that the change itself can impact trust relations.

Common for any trust management in the risk management setting is the dynamic and evolving nature of trust. For a given trust relation, the trust level, and thus the trust-based decision, might change over time, even for the same trustor, trustee, and action, because the trustworthiness evidence might change—for example, if the trustee acts

deceitfully or makes a severe mistake, or if the trustee's reputation changes.

Maintenance perspective

The basis for a trust assessment from the maintenance perspective lies in the previously conducted assessment, which might need updating to reflect changes that can, for instance, provide improved mechanisms for authentication and nonrepudiation that should relax requirements on the trustees' trustworthiness. Or it could be an increase in threats such as viruses and infected websites that should result in a stricter **trust policy**. From the maintenance perspective, the trust-based decision points are basically the same after the changes, but the previous assessments to evaluate trust and identify appropriate trust policies might no longer be valid. Changes in the level of potential gain and loss associated with a trust relation can also be affected.

Starting from the old target description and the old risk picture, the methodological challenges of the maintenance perspective involve facilitating a systematic reassessment of trust relations: for each change in the target or its environment, we must check whether any trust relation is affected and, if so, determine the effect on the target's factual risk level.

Before-after perspective

In the before-after perspective, the changes are planned or anticipated, so we can predict their effect on trust relations. Because the changes could be substantial, we might not only need to reassess existing trust relations but also consider that new relations can arise and old ones disappear. Such a change can, for example, be caused by an enterprise entering a joint venture with another, which could involve the exchange of sensitive information such as trade secrets and intellectual properties. The future decisions of whether to reveal certain information might then need to be based on trust relations.

The methodological challenges of the before-after perspective involve identifying the trust relations that persist through the changes and will therefore still remain, how to identify the trust relations that changed and therefore must be reassessed, how to identify and reassess the new trust relations from scratch, and how to identify the trust relations that must be removed. The challenges further involve assessing the impact of the change process itself on trust relations.

Continuous-evolution perspective

The continuous-evolution perspective addresses predictable changes, which can also involve alterations to trust relations and levels, as well as potential loss and gain. A continuous evolution could, for example, be the steady and predictable increase of viruses and infected websites yielding a corresponding decrease in the trust-

worthiness of websites generally. The evolution toward more sophisticated methods for cybercriminals to exploit sensitive information provides further proof that the consequences of trust breaches could become more severe over time. The methodological challenges of this perspective involve being able to capture evolution with respect to notions such as trust, subjective risk, and subjective opportunity for the actors within the target and, moreover, relating these to the evolution of the target's factual risk picture.

Improving risk assessment to take evolution into consideration raises new, strongly situation-dependent, methodological needs. Three particular situations lead to three distinct assessment scenarios—maintenance, before-after, and continuous-evolution—each requiring distinctive procedures.

The notion of trust management has yet to be as well-established as risk management. Still, the same scenarios apply when evolution is taken into account in trust management, but with additional challenges originating from trust's highly dynamic nature. ■

Acknowledgments

The work on which this article reports was partly funded by the EU IST Framework 7 projects SecureChange and MASTER, as well as the DIGIT-project (180052/S10) funded by the Research Council of Norway.

References

1. ISO 31000, *Risk Management: Principles and Guidelines*, Int'l Organization for Standardization, 2009.
2. ISO Guide 73, *Risk Management: Vocabulary*, Int'l Organization for Standardization, 2009.
3. C.J. Alberts and A.J. Dorofee, *OCTAVE Method Implementation Guide Version 2.0*, Software Eng. Inst., Carnegie Mellon Univ., June 2001.
4. S.A. Sherer, "Using Risk Analysis to Manage Software Maintenance," *J. Software Maintenance*, vol. 9, no. 6, 1997, pp. 345-364.
5. M.S. Lund, F. den Braber, and K. Stølen, "Maintaining Results from Security Assessments," *Proc. 7th European Conf. Software Maintenance and Reengineering (CSMR 03)*, IEEE CS Press, 2003, pp. 341-350.
6. S. Goel and V. Chen, "Can Business Process Reengineering Lead to Security Vulnerabilities: Analyzing the Reengineered Process," *Int'l J. Production Economics*, vol. 115, no. 1, 2008, pp. 104-112.
7. E. Lee, Y. Park, and J.G. Shin, "Large Engineering Project Risk Management Using a Bayesian Belief Network," *Expert Systems with Applications*, vol. 36, no. 3, 2009, pp. 5880-5887.
8. M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management," *Proc. IEEE Conf. Security and Privacy (SP 96)*, IEEE CS Press, 1996, pp. 164-173.

9. D. Gambetta, "Can We Trust Trust?" *Trust: Making and Breaking Cooperative Relations*, Dept. Sociology, Univ. of Oxford, 2000, pp. 213-237.
10. A. Jøsang, C. Keser, and T. Dimitrakos, "Can We Manage Trust?" *iTrust 2005*, LNCS 3477, Springer, 2005, pp. 93-107.
11. B. Solhaug, D. Elgesem, and K. Stølen, "Why Trust Is Not Proportional to Risk," *Proc. 2nd Int'l Conf. Availability, Reliability, and Security (ARES 07)*, IEEE CS Press, 2007, pp. 11-18.
12. A. Refsdal, B. Solhaug, and K. Stølen, "A UML-Based Method for the Development of Policies to Support Trust Management," *Proc. 2nd Joint iTrust and PST Conf. Privacy, Trust Management and Security (IFIPTM 08)*, vol. 263, Springer, 2008, pp. 33-49.

Mass Soldal Lund is a research scientist at SINTEF ICT. His research focuses on formal and semiformal specification techniques and languages, risk analysis and threat modeling, and model-based testing. Lund received a PhD

in informatics from the University of Oslo. Contact him at mass.s.lund@sintef.no.

Bjørnar Solhaug is a research scientist at SINTEF ICT. His research focuses on methods and languages for the modeling and analysis of systems with respect to security, risk, and trust. Solhaug received a PhD in information science from the University of Bergen. Contact him at bjornar.solhaug@sintef.no.

Ketil Stølen is a chief scientist at SINTEF ICT and a professor at the University of Oslo. His research focuses on model-based system development, security, risk assessment, trust management, and formal methods. Stølen received a PhD in computer science from the University of Manchester. Contact him at ketil.stolen@sintef.no.



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.

CALL FOR ARTICLES

Software for the Multiprocessor Desktop: Applications, Environments, Platforms

PUBLICATION: January/February 2011

SUBMISSION DEADLINE: 1 July 2010

Multicore processors, like Nehalem or Opteron, and many-core processors, like Larrabee or GeForce, are becoming a de facto standard for every new desktop PC. So, many developers will need to parallelize desktop applications, ranging from browsers and business applications to media processors and domain-specific applications. This is likely to result in the largest rewrite of software in the history of the desktop. To be successful, systematic engineering principles must be applied to parallelize these applications and environments.

This special issue seeks contributions introducing readers to multicore and manycore software engineering for desktop applications. It aims to present practical, relevant models, languages, and tools as well as exemplary experiences in parallelizing applications for these new desktop processors. The issue will also sketch out current challenges and exciting research frontiers.

We solicit original, previously unpublished articles on topics over the whole spectrum of software engineering in the context of desktop microprocessors, including multicore, manycore, or both.

POSSIBLE TOPICS INCLUDE

- How to make programming easier for average programmers
- Programming models, language extensions, and runtimes

- Design patterns, architectures, frameworks, and libraries
- Software reengineering/refactoring
- Software optimizations, performance tuning, and auto-tuning
- Testing, debugging, and verification
- Development environments and tools
- Surveys of software development tools
- Case studies of consumer application scenarios
- Industrial experience reports and case studies

QUESTIONS?

For more information about the focus, contact the guest editors:

- Victor Pankratius, University of Karlsruhe-KIT; pankratius@acm.org
- Wolfram Schulte, Microsoft Research; schulte@microsoft.com
- Kurt Keutzer, Univ. of California, Berkeley; keutzer@eecs.berkeley.edu

For the full call for papers: www.computer.org/software/cfp1 or www.multicore-systems.org/specialissue

For author guidelines:

www.computer.org/software/author.htm

For submission details:

software@computer.org

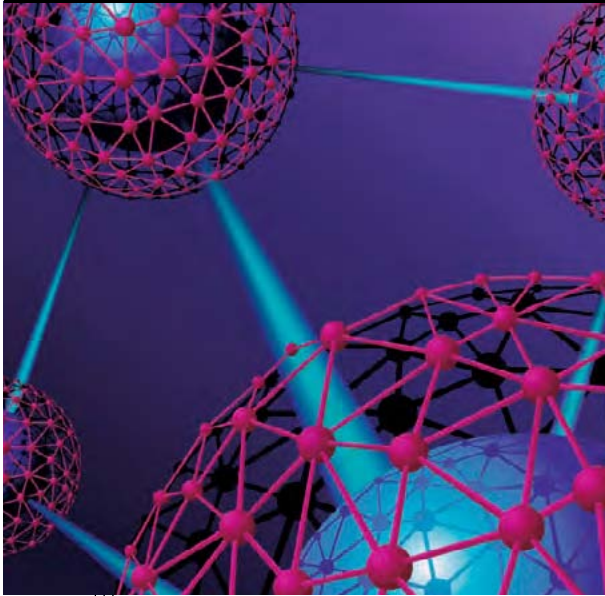
IEEE Software

Why Critical Systems Need Help to Evolve

by Bernard Cohen and Philip Boxer

Computer, vol. 43, no. 5, May 2010, pp. 56–63

DOI bookmark: <http://doi.ieeecomputersociety.org/10.1109/MC.2010.150>



WHY CRITICAL SYSTEMS NEED HELP TO EVOLVE

Bernard Cohen, *City University, London*
Philip Boxer, *Software Engineering Institute*

Classical engineering fails to model all the ways in which a critical sociotechnical system fits into a larger system. A study of orthotics clinics used projective analysis to better understand the clinics' role in a healthcare system and to identify risks to the clinics' evolution.

According to a 2009 report¹ on orthotic services in the UK, more than 1.2 million patients with conditions from diabetes to neuromuscular disorders rely on such services to enable them to work and live independently. In 2005, the report noted, it cost roughly £85 million (≈ US\$128 million) to provide orthotic services, and service demand had since been increasing commensurately with the aging population and the complexity of clinical conditions. Yet despite this increase, there appeared to be no consensus on how to relate the funding changes to the changing demand.

Given that early orthotic intervention improves lives and saves money, an orthotic-service provisioning system is certainly critical from the perspective of its patients. To manage its evolution, providers must understand the system's place within the larger system of National Health Services (NHS), and how it should respond to its patients' needs.

The 2009 report confirmed earlier findings² that for every £1 (≈ \$1.50) spent on orthotic services the NHS saves £4 (≈ \$6). With current expenditure on orthotic-service provisioning estimated at £100 million (≈ \$150 million), the NHS would save an estimated £400 million (≈ \$600 million). Nevertheless, the report found that, because of inadequate funding, pilot sites that had enhanced service levels could not sustain them. A hospital could implement recommendations only with specific funding from its Primary Care Trust. Moreover, increased awareness, not modeling, revealed the latent service demand, suggesting that current procurement practice is "too dependent on a commodity product procurement model."¹

Clearly, the report viewed the current operating environment of orthotic service providers as a threat to their ability to fulfill their mission. To improve patient care and provide real value to the NHS, the report recommended establishing a locally commissioned service based on clinical outcome. Such a solution is consistent with the 2008 Darzi report, which recommended transforming the NHS to a locally led, patient-centered, and clinically driven organization.³

Realizing this vision is not without challenges. Chief among them is the need to identify threats to the system, understand user demand patterns, and reach beyond classical engineering to adopt more appropriate modeling techniques for these more complex environments.

THREATS TO A SOCIOTECHNICAL SYSTEM

The threats facing any socio-technical system within a larger ecosystem such as the NHS extend beyond those of the familiar operational variety, where system components fail to perform as expected, individually or collectively. An orthotics service, for example, uses a model of how it should operate in providing orthoses to its patients. This model, in turn, determines how it actually operates.

Integral to an accurate system model is elaborating the distinction between “should operate” and “actually operates.” A fully elaborated model, such as that in Figure 1, should reflect three kinds of distinctions, or *cuts*: Cartesian, Heisenberg, and endo-exo.

Cartesian cut

Like the scientific method, engineering techniques rely on the successful construction of a modeling relation, as shown in the left side of Figure 1. A valid scientific theory is a formal system with an interpretation that maps the symbols in that system to observable states and events in a natural system in such a way that physical entailment (causality) in the natural system commutes with logical entailment (deduction) in the formal system. Engineers also rely on the existence of components whose composition into systems they can analyze—and occasionally synthesize—using the formal system’s calculus. Both science and engineering make the simplifying assumption that the natural systems they observe are *closed*, that is, immune to disturbance from all stimuli that the operative model does not account for. In other words, what you see is what you get.

However, unlike many systems, ecosystems are open because it is not possible to identify all the state components that some event does not alter. As such, these systems are exposed to the well-known frame problem.⁴

The distinction between what is and what is not accounted for by the observer’s knowledge is the observer’s Cartesian cut. The limitation is whether or not observers can assume that the system being modeled is closed. If the system is within an ecosystem, this assumption is invalid because *what you see is not what you get*.

In the context of orthotic services, the Cartesian cut presented a mismatch between the model of the clinic that defined its operational systems and the reality of its interactions with its patients and funders. The processes by which an orthotics service diagnoses particular patient

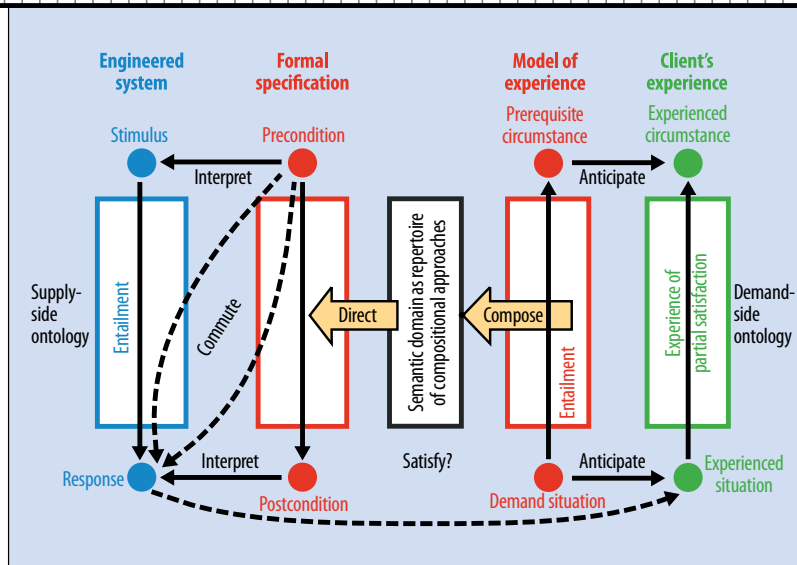


Figure 1. Fully elaborated modeling relation. A model that captures all the threats to a sociotechnical system must consider both the demand- and supply-side ontologies. Left: Construction of the modeling relation. Middle: Approaches used to compose the system, which the user must orchestrate. Right: Service demands on the basis of use context.

needs are affected by both how it is organized and how patients present their symptoms. Neither perspective can be defined wholly independently of the other.

Heisenberg cut

Collaboration across multiple sociotechnical systems—a system of systems (SoS)—raises the possibility that operationally adequate systems collectively behave in ways that violate their specifications. The sidebar “Defining a System of Systems” explains this behavioral characteristic in more detail. Even in a closed SoS, if analysts knew all the relevant compositional approaches (middle of Figure 1) but did not know the SoS’s behavioral domain, they would experience the SoS as open because its design did not fully determine its composition. Often, such systems engage in autonomous composition under the influence of user interactions, and their actual composite behavior differs from that interpreted from the composite model. In these instances, SoS behavior is considered emergent. An example of such behavior is when features interact in telecommunications systems.

The Heisenberg cut is the distinction between a system for which users can and cannot predict system behavior independently of their use of it. The limitation is whether or not observers can define the nature of the demands that a system is responding to independently of how the supplying system relates to those demands. For an ecosystem, it is impossible to make this assumption, since every observer is always also a participant within the ecosystem: Thus, *what you get depends on how you use it*.

➔ DEFINING A SYSTEM OF SYSTEMS

A directed system of systems (SoS) is treated as if it were still a single system, but its components have operational and managerial independence in the way they determine their respective behaviors.¹ A central authority predetermines the uses of these component systems, which is typically a universal ontological commitment as to what the system will be.

The integrated SoS is built and managed to fulfill specific purposes, such as air defense, to which the component systems' normal operational mode is subordinated. In practice, however, an SoS requires collaboration among its component systems concurrently with many other collaborations using the same systems. The agreed-upon central purpose thus depends on the way the component systems support these concurrent collaborations, which defer some ontological commitment to the time of use. Consequently, any centrally determined ontological commitment must underdetermine the component systems' uses. Central management organization cannot coerce the component systems, which are autonomous to the extent that they voluntarily collaborate to fulfill agreed-upon purposes. The Internet, for example, started out as directed, but its components can no longer be centrally managed.

In a virtual SoS—for example, an economy—there is not even a centrally agreed-upon purpose, so the component systems' support for the concurrent collaborations must rely on relatively invisible mechanisms (rules) to sustain the SoS.

Reference

1. M.W. Maier, "Architecting Principles for Systems-of-Systems," *Systems Eng.*, vol. 2, no. 1, 2009, pp. 267-284.

In the context of orthotic services, the Heisenberg cut was reflected in the underuse of orthoses relative to latent demand. The clinics measured demand in terms of acute episodes of care, rather than as multiple episodes of care within the context of a patient's chronic condition. An orthotics clinic is a practice that emerges from the composite effects of all its different parts interacting with aspects of its patients' lives and conditions. No observer, not even a participating observer, can wholly capture the nature of a clinic's practice. Any intervention must therefore take its place within the ongoing operational nature of that practice. A clinic cannot somehow stop and redesign itself, even though as a practice it can die.

Endo-exo cut

As expectations change, an individual system that meets its specification might fail to satisfy its users' demands when the system becomes part of an SoS. Exposure to these threats generates evolutionary pressures that require the system's stakeholders to understand its place within the SoS sufficiently to make strategic decisions that can mitigate those risks. The composite functionality that a collaborative SoS delivers is expressed as services composed by actors that are anticipatory systems⁵ within the larger ecosystem. These anticipatory systems define

service demands from their formulation of how those services affect their use context (right side of Figure 1).

Because these anticipatory systems are necessarily open, modeling their clients' needs also suffers from the frame problem. However, the system can model a client's need as an organization of demand that constitutes a pragmatics of use.⁶ That is, the client cannot know his needs directly, but can know them indirectly because he has experienced their effects.

The client's endo-exo cut is the difference between what the client can and cannot know directly about his needs. This distinguishes the knowledge that is implicit in a sociotechnical system's behavior (ontic knowledge) from what those observing the system can know about it (epistemic knowledge).⁷ For example, the behavior of a sociotechnical system is a result of both how it endogenously chooses to interact with its clients and how the design of its systems exogenously constrains it. This cut is a consequence of attributing agency to the sociotechnical system.

The limitation is whether or not service providers can grasp the full nature of the underlying reality, in which anticipatory processes are unfolding. In the context of the ongoing interactions within an ecosystem, such a full grasp is never possible: *What is wanted is never exactly what is asked for.*

In the context of orthotics services, the endo-exo cut reflects the failure of the larger healthcare ecosystem to evolve compatibly with a model of the clinic concerned with managing the lifelong development of a patient's condition.

MODELING A SOCIOTECHNICAL SYSTEM

Classical engineering is limited because it is impossible to fully separate any sociotechnical system from its context of use within an ecosystem. However, by enabling the members of and stakeholders in the sociotechnical system to analyze and project their participation experience, it is possible to understand how the sociotechnical system is defined in terms of the Cartesian, Heisenberg, and endo-exo cuts.

The techniques and tools of projective analysis facilitate this understanding, and support members and stakeholders in formulating and evaluating alternative evolutionary strategies with respect to the larger ecosystem. In the orthotics case, we used PAN,⁸ a particular implementation of projective analysis.

Modeling a client enterprise as a sociotechnical system requires accepting that the observer's perspective is always exogenous to the system, which is why any modeling is always a projection of the observer's model of the system in and of itself. For example, to work with the orthotics clinics, we had to model the way the clinics worked from the point of view of the clinicians and managers. Likewise, to understand how doctors and specialists refer patients

to the clinics, we had to model the referral pathways used by clinicians in the larger system.

Relationships among the cuts

The model must be able to account for the three cuts that the system makes in defining itself. As Figure 2 shows, the relationships among these cuts are in terms of a behavior domain and four quadrants that layer the client's relationship to demand: what, how, for whom, and why. The behavior domain comprises the kinds of behavior that define the client system and its customer interactions: for example, the clinical orthotic practices and the contexts for engaging in them.

What. This perspective reflects what the clinic does, or the material nature of the clinic's work, as in what an orthotist actually does. As such, it describes the clinic as a system in terms of its realized behavior: what critical technologies it has mastered and the source of its products or services (constituent performances). The *what* perspective might be an observation of the way the overall clinic functions day to day, for example.

How. This perspective identifies the clinic's characteristics: What makes a clinic unique? What organizational aspects define that clinic's identity, such as how a clinic organizes its work to be effective? This perspective describes the clinic's authorized models. It looks at the key constituent performances it needs to construct the output performances it provides to its patients (customers), such as understanding how the clinic is actually organized.

For whom. This perspective clarifies whom the clinic is serving and identifies the economics of this service, such as the specific conditions the orthotics clinic is treating. This perspective also describes the patients' demands in the clinical environment. How must the clinic customize and orchestrate its outputs to generate the composite capabilities its patients need for their particular situations, and how will the clinic synchronize these composite capabilities with those situations? An example is seeking to understand how clinics actually apply orthotic treatments within the context of their patients' daily lives.

Why. This perspective looks at what makes the clinic's identity-defining characteristics of value within the NHS, particularly in relation to its patients. That is, what in the NHS drives the clinic's value, such as what is the larger context of the patient's life and condition that is giving rise to the presenting symptoms? This perspective also describes the environmental models that prompt demand. What use context is generating the demand that the clinic

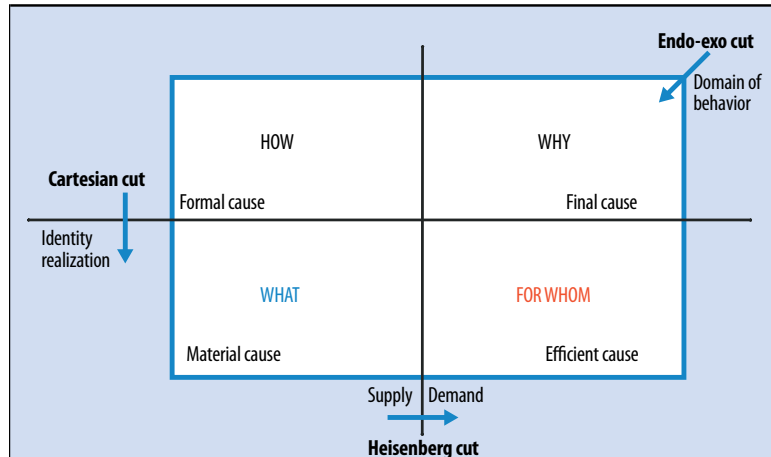


Figure 2. Modeling the Cartesian, Heisenberg, and endo-exo cuts and their interrelationships partitions the behavior domain into four quadrants. These four quadrants—what, how, for whom, and why—stratify the client's relationship to demand.

is targeting, and what is driving that context? For example, this perspective might bring to light the characteristics of the NHS and patient environment in which the clinic's practice is situated.

Identifying asymmetries

The stratified relationships among cuts also underline three asymmetries that must be addressed if the client is to manage its relationship to changes in its demand environment. The *what* and *how* perspectives span the first asymmetry: *The technology does not define the product*. The ability to manage the technology generates economies of scale in production. The manufacturing methods per se should not define how clinics use orthoses to treat patients.

The *how* and *for whom* perspectives span a second asymmetry: *The business model does not define the customer's solution*. The ability to manage the business model generates economies of scope in the various markets that can be served, but the ways in which the clinic organizes its treatment process should not define what treatments it can provide particular patients.

The *for whom* and *why* perspectives span the last asymmetry: *The patient's demand does not define the experience that the patient wants*. The ability to manage the relationship to demand generates economies of alignment in the way the customer's experience is supported. For example, the demands of the symptoms in a single episode should not define the larger multi-episode treatment strategy that a patient might need throughout the condition's life.

The first two asymmetries assume that providers can define the demand environment to be independent of the client enterprise's behavior. The classical engineering disciplines are therefore well suited to mitigating the threats

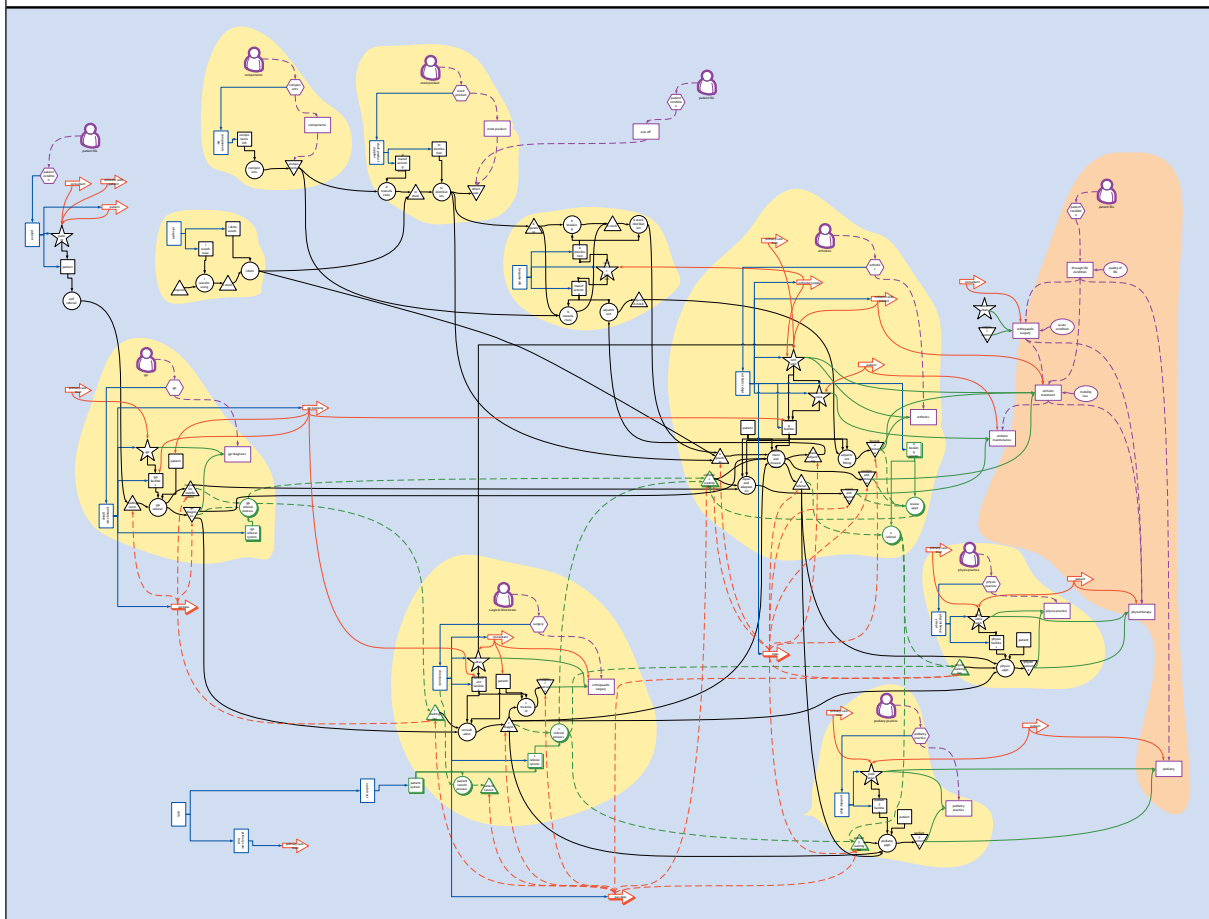


Figure 3. Model for orthotics services that combines the *what*, *how*, *for whom*, and *why* perspectives in the context of orthotic services. The model is a layered graph, with each layer corresponding to the structure, function, hierarchy, synchronization, information, and demand of the client enterprise, which in this case is the clinic. The colored regions represent clinical functions (such as orthopedic surgery and outpatient services), patients' conditions, and supplier services.

that arise in these environments. The third asymmetry, however, places the client enterprise explicitly within a dynamic ecosystem. A client enterprise that fails to comprehend and accommodate itself to this will limit its possible competitive behaviors, exposing itself to threats created by the changing nature of demand inherent in an ecosystem. It is these threats that a model based on all four perspectives can locate and identify.

PROJECTIVE ANALYSIS OF ORTHOTICS CLINICS

Figure 3 shows the model we elicited for orthotics services using Visual PAN, an application of Microsoft Visio with a customized stencil. The model is in the form of a layered graph, with each layer corresponding to an aspect of the clinic that several perspectives share.

This graph is effectively a heterogeneous binary relation that PAN tools can manipulate algebraically to generate the

dependency structure matrix (DSM) in Figure 4. This structure was the basis for the commodity product procurement focus identified in the 2009 survey of orthotic services.¹ Figure 5 illustrates the stratification matrix, which is more complete and hence much more complex.

As the DSM and stratification matrix show, the complexity of managing the third asymmetry—aligning the ability to generate treatment with the patient's particular needs—overshadows the relative simplicity of the underlying activities.

Using an extended form of Q-analysis,⁹ an analyst can generate 3D histograms, or landscapes, from selected submatrices of the stratification matrix. Figure 6 shows a landscape for the orthotics services system showing the relationships among major organizational components.

We also analyzed the roles of the clinics' various data platforms. Figure 7 shows the landscape for this analysis. Although the platforms overlapped on appointment

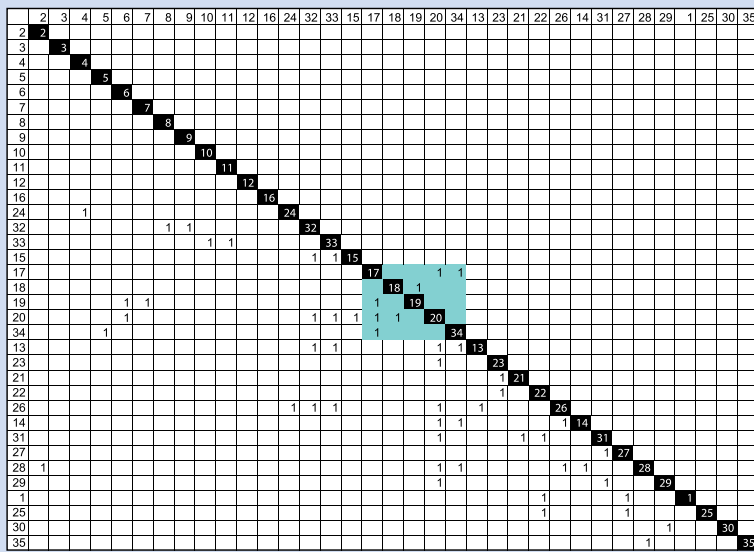


Figure 4. Dependency structure matrix for the model in Figure 3. The DSM captures only one of the aspects in the model, showing a relatively simple supply structure with some feedback relationships (blue box) around the actual orthoses fitting. The names of the rows and columns (not shown for simplicity) are the processes derived from the fully elaborated model.

and patient details, all the clinical data relevant to the particular patient condition were held in separate, unrelated silos.

The projective analysis supported several actions and interventions that significantly improved orthotic clinics' ability to deliver quality care. Not the least of these was the need to support the alignment processes themselves. According to the original 2004 survey of pathfinder clinics,² no clinic reported outputs by episode or analyzed referral by condition. The only reporting was on the clinic's cost, and "even this was generally poor." As a result, clinics had no shared experience reports or information base to help them improve operations or justify any investment. The report also noted the lack of data related to the chronic nature of the conditions being treated. In addition to the inac-

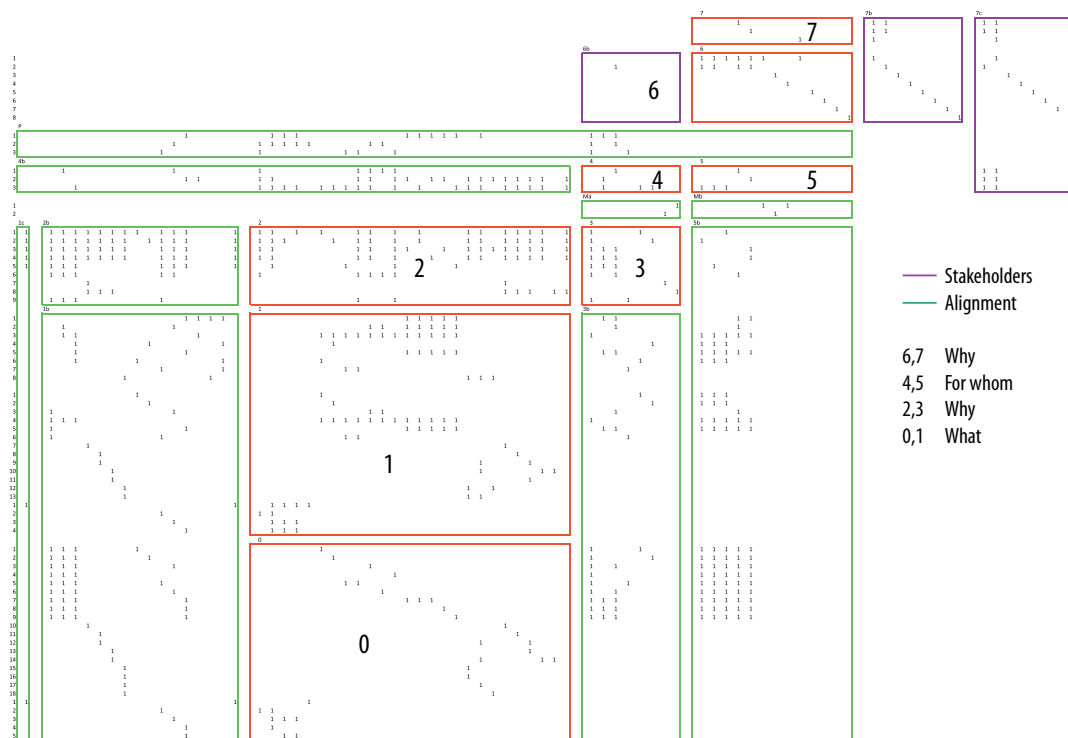


Figure 5. Stratification matrix for the model in Figure 3. This matrix is much more complete and thus much more complex than the DSM in Figure 4. The red matrices correspond to the stratification, the mauve matrices show the stakeholder influence, and the green matrices show how the DSM activities align to patient demands. The names of the rows and columns (not shown for simplicity) are events and processes, respectively, both of which are derived from the fully elaborated model.

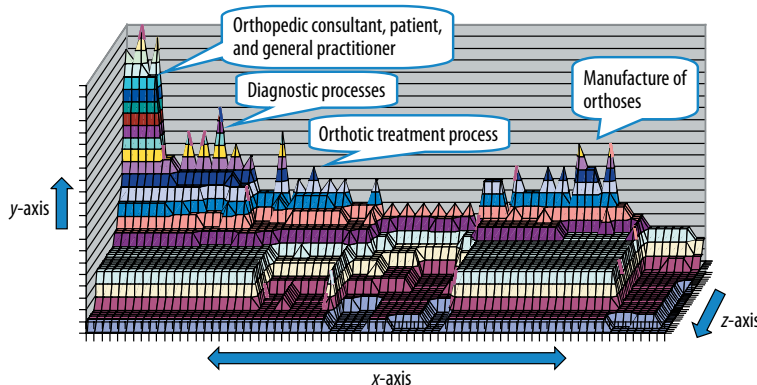


Figure 6. Cross-sectional landscape. A landscape shows gaps in the relationships between the major components of the organization being modeled, revealing the risks to stakeholders. The component outputs (not shown for simplicity) are ordered along the x-axis, the y-axis shows the complexity of alignment behind each output, and the z-axis shows the extent of overlapping complexity between outputs. The peaks represent areas of alignment that must themselves be aligned by social processes within the ecosystem as a whole.

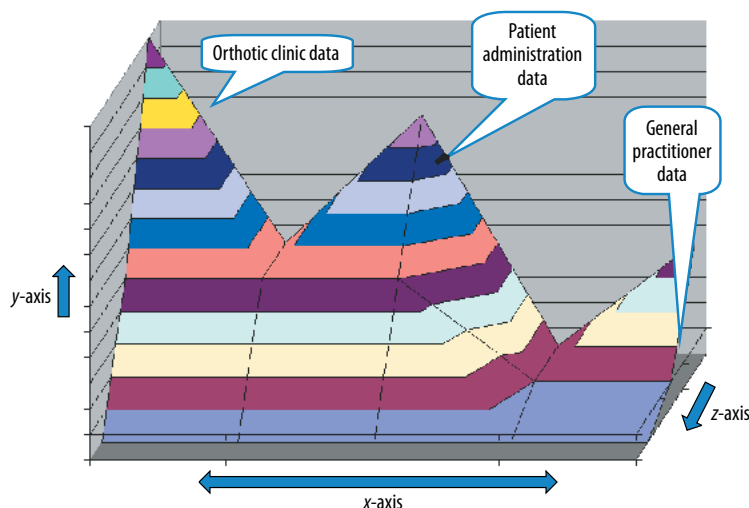


Figure 7. Data platform landscape. The data platforms are ordered along the x-axis, the y-axis shows the number of data elements synchronized by platform, and the z-axis shows the number of platforms with this synchronization level.

cessibility of patient records, the report found holes in the data on the conditions that defined an episode, on referral pathways, and on episode characteristics.

The application of projective analysis to orthotics clinics revealed the complexity of the alignment processes needed to deliver effective care to their patients. It also identified holes in the data being collected—gaps that not only prevented the clinics from acting in the most efficient and effective

way, but that also kept the larger health-care system from attaching value to a changed way of clinical operation. One of the recommendations made, therefore, was to have the clinics deploy a data platform to pull the missing information as it was generated and make it available for the stakeholders in their subsequent decision making.²

However, given its other funding priorities, the NHS rejected the proposed transformation of the clinics on cost grounds, despite the evidence that the returns in efficiency and patient care would be roughly four times the investment. Why should there be such an obstacle to this critical system's evolution?

At first glance, the recommendation to deploy a data platform seems similar to a recommendation for any traditional information systems requirements analysis. However, the data platform was a by-product of our analysis, not its primary objective. From the perspective of the clinics' role, deploying the data platform would have seriously affected the NHS's trust structure and the centralized patient record system that it was installing. The obstacle was therefore at a much higher level of understanding—that of the ecosystem itself and its reluctance to address the consequences of the third asymmetry.

Requirements analysts have often reported similar results, considering them merely exceptions to an otherwise classical engineering analysis. We suggest that, as they evolve, critical systems are inevitably exposed to higher-order risks, which classical engineering methods fail to identify. Projective analysis offers a more cost-effective alternative. **□**

References

1. J. Hutton and M. Hurry, "Orthotic Service in the NHS: Improving Service Provision," *Proc. York Health Economics Consortium, Univ. of York*, July 2009; <http://www.bapo.org/docs/latest/york%20report.pdf>.
2. T. Flynn and P. Boxer, "Orthotic Pathfinder Report," *Business Solutions Ltd.*, July 2004, pp. 60-75.
3. Lord Darzi, "High Quality Care for All: NHS Next Stage Review Final Report," UK Dept. of Health, June 2008.

4. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," *Machine Intelligence*, vol. 4, Edinburgh Univ. Press, 1969, pp. 463-502.
5. R. Rosen, *Life Itself*, Columbia Univ. Press, 1985.
6. C.S. Peirce, "How to Make Our Ideas Clear," *Popular Science Monthly*, Jan. 1878; <http://www.peirce.org/writings/p119.html>.
7. H. Atmanspacher, "Exophysics, Endophysics, and Beyond," *Int'l J. Computing Anticipatory Systems*, vol. 2, 1998, pp. 105-114.
8. W. Anderson and P. Boxer, "Modeling and Analysis of Interoperability Risk in Systems of Systems Environments," *CrossTalk*, Nov. 2008; <http://www.stsc.hill.af.mil/crossTalk/2008/11/0811AndersonBoxer.html>.
9. R.H. Atkin, "The Methodology of Q-Analysis: How to Study Corporations by Using Concepts of Connectivity," *Management Decision*, vol. 18, no. 7, 1993, pp. 380-390.

Bernard Cohen is an honorary visiting professor in the School of Informatics at City University, London. His research interests span the gaps between programming practices, formal computer science, and human agency. Cohen received a BSc in natural philosophy and a post-

graduate diploma in numerical analysis and computer programming, both from Glasgow University. He is a chartered engineer, a Fellow of the British Computer Society, and a member of the Institution of Engineering and Technology. Contact him at b.cohen@city.ac.uk.

Philip Boxer is a senior member of the technical staff at the Software Engineering Institute of Carnegie Mellon University. His research interests include the economics and architectural and risk characteristics of the sociotechnical ecosystems within ultra-large-scale systems. Boxer received a BSc in electrical and electronic engineering from King's College in London University and an MSc in business administration from the London Graduate School of Business Studies. He is a member of IEEE, the International Council on Systems Engineering, and the Institute of Business Consulting. Contact him at pboxer@sei.cmu.edu.



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.



Computer
magazine
looks ahead
to future
technologies

IEEE

IEEE
computer
society

Computer

Welcomes Your Contribution

- Computer**, the flagship publication of the IEEE Computer Society, publishes peer-reviewed technical content that covers all aspects of computer science, computer engineering, technology, and applications.
- Articles selected for publication in **Computer** are edited to enhance readability for the nearly 100,000 computing professionals who receive this monthly magazine.
- Readers depend on **Computer** to provide current, unbiased, thoroughly researched information on the newest directions in computing technology.

To submit a manuscript for peer review, see **Computer's** author guidelines:

www.computer.org/computer/author.htm

Simplicity as a Driver for Agile Innovation

by Tiziana Margaria and Bernhard Steffen

Computer, vol. 43, no. 6, June 2010, pp. 90–92

DOI bookmark: <http://doi.ieeecomputersociety.org/10.1109/MC.2010.177>

Simplicity as a Driver for Agile Innovation

➤ **Tiziana Margaria**, *Potsdam University*

➤ **Bernhard Steffen**, *TU Dortmund University*



Software and hardware vendors long avoided interoperation for fear of opting out of their own product lines. Yet decisive change came to the automobile industry from a holistic evolution and maturation on many fronts.

Looking at software system production and use today, we can easily compare the industry's current life cycle to that experienced by the automobile industry 80 years ago. The following statement, attributed to Gottlieb Daimler, characterizes car-makers' expectations at that time: "The market for automobiles will never grow beyond one million cars, for a very simple reason: Who would educate all those chauffeurs?"

This skepticism is understandable—back then, cars were handcrafted and cost more than a house. At the time, they were technically amazing—they could go up to 100 kph—but they had a hefty downside—the mean distance between flat tires averaged 30 km thanks to nail damage from horses and carts.

Not surprisingly, the number of extra tires constituted a status symbol: two full wheels were normal, with some cars carrying up to eight extra wheels to weather longer trips. But those who could afford a car were neither willing to change tires nor eager to maintain the engine, making well-trained chauffeurs an indispensable commodity in the 1920s.

So it goes with software. Despite the promises and effort, working

with software products still offers a comparable adventure, one that rarely proceeds as expected. Difficulties with deployment and use lead to enormous system, organizational, and personal performance losses, not only at first deployment but even more so when we factor in the inevitable upgrades, migrations, and version changes.

THE PRICE FOR THE PACE

Millions of users suffer when standard software with a large market share evolves. Maybe it undergoes a radical redesign of the graphical user interface (GUI) or offers a new generation of tools not readily compatible with previous versions. Users must then desperately search for previously well-understood functionality, spending hours or even days bringing perfectly designed documents to a satisfactory state within this changed technical environment.

This frustrating catch-up phase causes an enormous productivity loss that can force customers to shy away from updates and migrations, sticking instead with old and even outdated or discontinued products or versions. In many situations, customers fear any kind of innovation involving IT because they immediately associate

a change with enormous disruptions and long periods of instability. With technology-driven innovations, this fear is justified thanks to the new technologies themselves. However, even small and technically simple adaptations to a business process typically require a major IT project, with all its involved risks.

Thus, decision makers act conservatively, preferring patches and exchanging functionality only when it's absolutely necessary. Even the automobile industry fails when it comes to IT adoption and, particularly, IT agility. Much of a car's control software runs on specific hardware, which limits the software's applicability, especially after the hardware becomes obsolete—the software can't be ported elsewhere, meaning the manufacturer is more or less stuck with that hardware.

It takes engineers years to innovate, which the product life cycle then outlives by factors beyond that of the electronics and software within. The central problem is the IT lock-in at design time: decisions on which technology to use and long-term deals with the manufacturers are frozen before production starts and often last beyond the facelifts that periodically refresh these products.

In the aerospace industry, this life-time mismatch is even more evident: it takes decades to plan and design a mission, which leaves the IT used in the field in a typically decades-old state. IT innovation is the fastest we observe, and it systematically outpaces the life cycle of the products built using it. Inevitably, the products' life spans shorten to those of the IT they embody, as in consumer electronics, but this is unacceptable for expensive products.

Today, we have a similar situation in IT: singularly taken, the technologies and products are well-designed and innovative, but aren't made for working together and can't evolve independently. Consequently, we work with systems whose stability isn't proven and in which we can thus pose only limited trust. Once a bearable situation is achieved, and a constellation works, we tend to stick to it, bending the business and procedures to fit the working system, then running it until support is discontinued, if then. This shows that even pure software-based IT is often caught in the platform lock-in trap: business needs too often outpace the life cycle of the IT platforms that steer a company's organization and production.

STATES OF THE ART

Various factors contributed to our current state of the art. Some are rooted in the business models of major software and hardware vendors, who long avoided interoperability for fear the consequences of opting out of their own product lines would be dire. The frantic pace of technology provides its own chaos: before a certain technology reaches maturity and can repay the enormous investments for its development and production, a newer option attracts attention with novelty and fresh promises.

Decisive change came to the automobile industry not from the isolated improvement of single elements but

from a holistic evolution and maturation on many fronts, with the interplay of numerous factors:

- *Better, more robust components.* The modern car platform approach builds on comparatively few well-engineered individual components, such as the tires, motor, and the chassis.
- *Better streets.* Today, we hardly need worry about flat tires.

Even pure software-based IT is often caught in the platform lock-in trap.

- *Better driving comfort.* Cars run smoothly, reliably, and safely, even if maltreated. User orientation has made a huge difference: drivers don't need to be mechanics.
- *Better production processes.* Modern construction supports cars tailored to their customers, even if all are built on platforms. Essentially, no two delivered cars are identical, but all are bound to only a few well-developed platforms.
- *Better maintenance and support.* Drivers have access to support worldwide, which can even include home transportation.

These modern developments have a straightforward match to the situation in IT, while also revealing the weaknesses of today's IT industry:

- *Better, more robust components.* Today's components are typically too complicated and fragile, and therefore are difficult to integrate in larger contexts. Service orientation seems to be a potentially strong step in the right direction, but it must be combined with a clear policy.
- *Better connection and interop-*

eration. We still lack seamless connection and integration, with numerous mismatches at the protocol, interface, or behavioral level. Meanwhile, the intended semantics and accompanying security provide an everlasting concern and a hot research topic.

- *Better user comfort.* Experts might know various specifically optimized solutions, but normal users find none. Even getting a modern phone to simply make a call can be rather frustrating, with many perceived extra steps and commands.
- *Better production processes.* Application development and quality assurance should be directly steered by user requirements, controlled via user experience, and continuously subject to modification during development.
- *Better maintenance and support.* Established scenarios and often-used functionality should continue to work, while support should be immediate and integrated into the normal workflow.

The transition to overcoming these weaknesses will depend on adopting economical principles that favor dimensions of maturity and simplicity over sheer novelty. In our analogy, Formula One car racing is an attractive platform for high-end research, but is unsuited for the needs and requirements of mass driving due to different skills, costs, and traffic conditions. Taking ideas and results from the high-end and specialized laboratory product requires diverse and extensive research to succeed. Transferred to the IT domain, this kind of research spans several dimensions:

- *Human-computer interaction* has led to GUIs that provide an intuitive user interface.
- *Domain modeling and semantic technologies* can establish a

user-level understanding of the involved entities.

- *Cloud computing* and other forms of platform virtualization provide stable user-level access to functionality.
- *Service orientation and process technologies* offer easy interactive control at the user process level.
- *Integrated product line management and quality assurance* requires validation and monitoring to guarantee correctness criteria at design, orchestration, and runtime.
- *Rule-based control* helps developers react flexibly to unforeseen situations.
- *Security and safety* affect not only business-critical applications but also technologies for establishing a high level of fault tolerance, be it at the infrastructural, software, or human level.
- *Major application domains*, such as business, biology, or medicine, keep the focus on constant awareness of the primary issue—user requirements.

The contributions of these individual research areas must be combined holistically to successfully control, adapt, and evolve systems composed of mature components.

THE PRICE FOR MATURITY

Achieving a sufficient level of maturity across components, connections, interoperation, and evolution is a complex and highly interdisciplinary task that requires technological knowledge and deep domain modeling expertise.

In this setting, standard investigation topics in IT such as complex architectural design and computational complexity are only of secondary and ancillary importance. The key to success is application of the “less is more” principle, with the goal of treating simple things simply, by a correspondingly simple design reminiscent of Lego blocks: primitive and well-defined blocks combine to

reliably create complex solutions.

Developers might argue that there is no universal approach, but several domain-, purpose-, and profile-specific approaches within their scope are possible that capture the vastness of today’s programming problems much more simply, reliably, and economically than most people think. This approach trades generality, which must be complex to accommodate diverse and sometimes antagonistic needs, with simplicity.

Companies such as Apple have successfully adopted simplicity as a fundamental design principle—for example, insights that simplify its users’ lives concern both the handling of its products and their maintenance and robustness. Users adopted these innovations enthusiastically and pay a premium price for this “IT simply works” experience. Similarly, Windows 7 attempts to overcome the tendency to provide cutting-edge and increasingly complicated technology in favor of a more user-driven philosophy. Combining extensive interviews and agile methods in its development accelerated this paradigm shift.

While promising beginnings, these initiatives fall short of making mature technologies that simply work a widespread reality. We need extensive research and a clear engineering approach tailored to simplicity.

IT SIMPLY WORKS

The potential of a slogan like “IT simply works” offers vast opportunities unrestrained by the physical limitations of classical engineering. In principle, every software component can be exchanged at any time, almost everywhere, without leaving any waste—an ideal situation for truly component-based engineering.


Leveraging this potential would economically surpass the impact of producing new products based on leading-edge IT. Studies of product innovation show that technological leadership corresponds only to a relatively small fraction of market

success and new market creation.

Most often, technology-driven innovation accompanies risk caused by the new technologies themselves. Innovations rooted in the *business purpose*, such as the service to the user or customer, have a much higher chance of success because user-level advantages are easier to communicate in the market, especially if detached from technological risks.

Improved levels of maturity can enable a new culture of innovation on the application side. Once we overcome the fear of change, true agility will guide the application experts, leading to new business models and new markets. History shows that with the availability of reliable cars, totally new forms of transportation and business arose.

For the software industry, maturity could revolutionize software’s mass construction and mass customization far beyond our experience in the automotive industry. Theoretically, we can easily “change wheels while driving” and decompose and reassemble the entire car or bring new passengers aboard at the speed of light without being bound to specific hardware.

From a higher perspective, drawing adequate lines here can be considered a distinguishing trait for this new line of research and play a central role in the evolution of our economy and society. 

Tiziana Margaria is chair of service and software engineering at the Institute of Informatics, Potsdam University, Germany. Contact her at margaria@cs.uni-potsdam.de.

Bernhard Steffen is chair of programming systems in the Department of Computer Science, TU Dortmund University, Germany. Contact him at steffen@cs.tu-dortmund.de.

Editor: Mike Hinchey, Lero—The Irish Software Engineering Research Centre; mike.hinchey@lero.ie

RECOMMENDED RESOURCES

0. M. Hinchey and L. Coyle, *Evolving Critical Systems*, tech. report Lero-TR-2009-00, Lero—the Irish Software Engineering Research Centre, July 2009 (and revisions), <http://www.lero.ie/ecs/whitepaper>.

A manifesto for an ECS research agenda, regularly updated, and on which some of this material is based.

1. E.B. Swanson, “The dimensions of maintenance,” *Proceedings of the 2nd International Conference on Software Engineering* (ICSE 1976), IEEE Computer Society Press, 1976, pp 492–497.

A classic paper that describes the various types of software maintenance, classifying them as *corrective*, *perfective*, or *adaptive*.

2. V. Rajlich and K.H. Bennett, “A Staged Model for the Software Life Cycle,” *Computer*, vol. 33, no. 7, July 2000, pp. 66–71.

Presents a staged life cycle model highlighting the maturity of a software system as being an essential consideration when planning change.

3. P. Naur and B. Randell, eds., *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968*, Scientific Affairs Division, NATO, Brussels, 1968.

A now-famous account of a NATO workshop where the term “software engineering” is believed to have been first coined. The report places much emphasis on both evolution and criticality of software.

4. M.M. Lehman, “Laws of software evolution revisited,” *Proceedings of the 5th European Workshop on Software Process Technology* (EWSPT 1996), Springer-Verlag, 1996, pp. 108–124.

A revised version of Lehman’s now-famous “Laws of Software Evolution,” which considers the laws’ relevance and adds additional laws, making it the first complete description.

5. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, “Towards a taxonomy of software change,” *Journal of Software Maintenance and Evolution*, vol. 17, no. 5, Sep. 2005, pp. 309–332.

Provides a taxonomy of different reasons for and approaches to evolving software.

6. M.R. Lyu, ed., *Handbook of Software Reliability and System Reliability*, McGraw-Hill, 1996.

An excellent and much-cited reference on critical systems and software reliability.

7. N.G. Leveson, “Software safety: why, what, and how,” *ACM Computing Surveys*, vol. 18, no. 2, June 1986, pp. 125–163.

A comprehensive overview of software safety, with particular reference to the issues that distinguish safety in software from safety in other domains.

ABOUT THE EDITORS

Mike Hinchey is Director of Lero—the Irish Software Engineering Research Centre and Professor of Software Engineering at University of Limerick, Ireland. Prior to joining Lero, Hinchey was Director of the NASA Software Engineering Laboratory; he continues to serve as a NASA Expert. In 2009 he was awarded NASA’s Kerley Award as Innovator of the Year.

Hinchey holds a B.Sc. in Computer Systems from University of Limerick, an M.Sc. in Computation from University of Oxford, and a PhD in Computer Science from University of Cambridge. The author/editor of more than 15 books and over 200 articles on various aspects of Software Engineering, Hinchey is Chair of the IFIP Technical Assembly and Chair of IFIP Technical Committee 1 (Foundations of Computer Science), Chair of the IEEE Technical Committee on Complexity in Computing, as well as Editor-in-Chief of *Innovations in Systems and Software Engineering: a NASA Journal* (Springer) and AIAA’s *Journal of Aerospace Computing, Information, and Communication*. Contact him at mike.hinchey@lero.ie.



Lorcan Coyle is a research fellow with Lero—the Irish Software Engineering Research Centre at University of Limerick, Ireland. Prior to joining Lero, Coyle was a postdoctoral researcher with the Systems Research Group in University College Dublin, Ireland.

Coyle holds a Bachelor’s Engineering degree and a PhD in Computer Science from Trinity College Dublin. He is a member of the Irish Computer Society and Engineers Ireland. He has chaired the 20th Irish Conference on Artificial Intelligence and Cognitive Science. Coyle’s research interests include Software Engineering, Autonomic Computing, Context-Aware Systems, Pervasive and Ubiquitous Computing, and Machine Learning. He has authored more than fifty publications at various fora, including *Computer*, *PerCom*, *Knowledge Engineering Review*, *ICPS*, *EuroSSC*, *IUI*, and *CACM*. Contact him at lorcan.coyle@lero.ie.



ABOUT THE EDITORS

Bashar Nuseibeh is a Professor of Computing at The Open University (Director of Research, 2002-2008) and a Professor of Software Engineering and Chief Scientist at Lero—the Irish Software Engineering Research Centre. His research interests are in software requirements and design, security and privacy, process modelling and technology, and technology transfer.

Nuseibeh has consulted widely with industry, working with organizations such as the UK National Air Traffic Services (NATS), Texas Instruments, Praxis Critical Systems, Philips Research Labs, and NASA. His work included successful analysis of software requirements of the International Space Station at NASA, analysis of aircraft conflict alert and security analysis of new technologies at NATS, and feature evolution analysis of the SPARC compiler at Praxis. He currently serves as Editor-in-Chief, *IEEE Transactions on Software Engineering*. Contact him at B.Nuseibeh@open.ac.uk.



José Fiadeiro is Professor of Software Science and Engineering and Head of the Department of Computer Science at the University of Leicester, United Kingdom. He did his undergraduate degree in Mathematics at the University of Lisbon (Faculty of Science), after which he moved to the Technical University of Lisbon (Department of Mathematics, Faculty of Engineering), where he studied for a PhD under the supervision of Amílcar Sernadas. He was awarded a doctorate in 1989 and then spent three years doing research at Imperial College London with a grant from the European Commission. He became Associate Professor in Computer Science at the Technical University of Lisbon in 1992 and moved to the University of Lisbon (Department of Informatics, Faculty of Science) in 1993. Before he joined Leicester in 2002, he held visiting research positions at Imperial College, King's College London, PUC–Rio de Janeiro, and the SRI International. He became Head of Department at Leicester in August 2006 and is also a Fellow of the British Computer Society. Contact him at jose@mcs.le.ac.uk.

