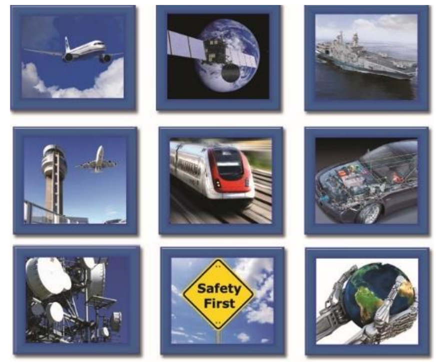


22 February 2018

Automotive Hot Topics

The IEEE Software Special Issue on Automotive Software

John Favaro

john.favaro@intecs.it

MAY/JUNE 2017
VOL. 34, NO. 3

IEEE Software

AUTOMOTIVE SOFTWARE

Automotive Software

Christof Ebert, Vector Consulting Services

John Favaro, Intecs

SOFTWARE IS THE number-one decisive competitive factor in the automotive industry. Innovations such as driver-assistance systems and energy-efficient driving require co plex solutions with complex s ware functionality. Not only m the growing complexity be manag

intecs Solutions
the Brainware company

AUTOMOTIVE SPIN ITALIA

Deep Learning in Automotive Software

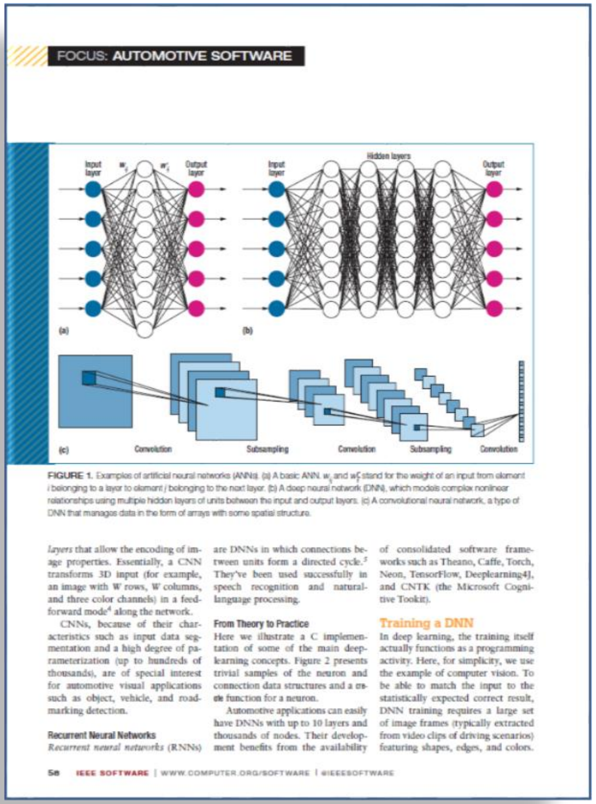

FOCUS: AUTOMOTIVE SOFTWARE

FIGURE 1. Examples of artificial neural networks (ANNs). (a) A basic ANN. w_i and w_i' stand for the weight of an input from element i belonging to a layer to element j belonging to the next layer. (b) A deep neural network (DNN), which models complex nonlinear relationships using multiple hidden layers of units between the input and output layers. (c) A convolutional neural network, a type of DNN that manages data in the form of arrays with some spatial structure.

layers that allow the encoding of image properties. Essentially, a CNN transforms 3D input (for example, an image with W rows, W columns, and three color channels) in a feed-forward mode[4] along the network.

CNNs, because of their characteristics such as input data segmentation and a high degree of parameterization (up to hundreds of thousands), are of special interest for automotive visual applications such as object, vehicle, and road-marking detection.

Recurrent Neural Networks

Recurrent neural networks (RNNs) are DNNs in which connections between units form a directed cycle.[5] They've been used successfully in speech recognition and natural-language processing.

From Theory to Practice

Here we illustrate a C implementation of some of the main deep-learning concepts. Figure 2 presents trivial samples of the neuron and connection data structures and a create function for a neuron.

Automotive applications can easily have DNNs with up to 10 layers and thousands of nodes. Their development benefits from the availability of consolidated software frameworks such as Theano, Caffe, Torch, Neon, TensorFlow, Deeplearning4J, and CNTK (the Microsoft Cognitive Tookit).

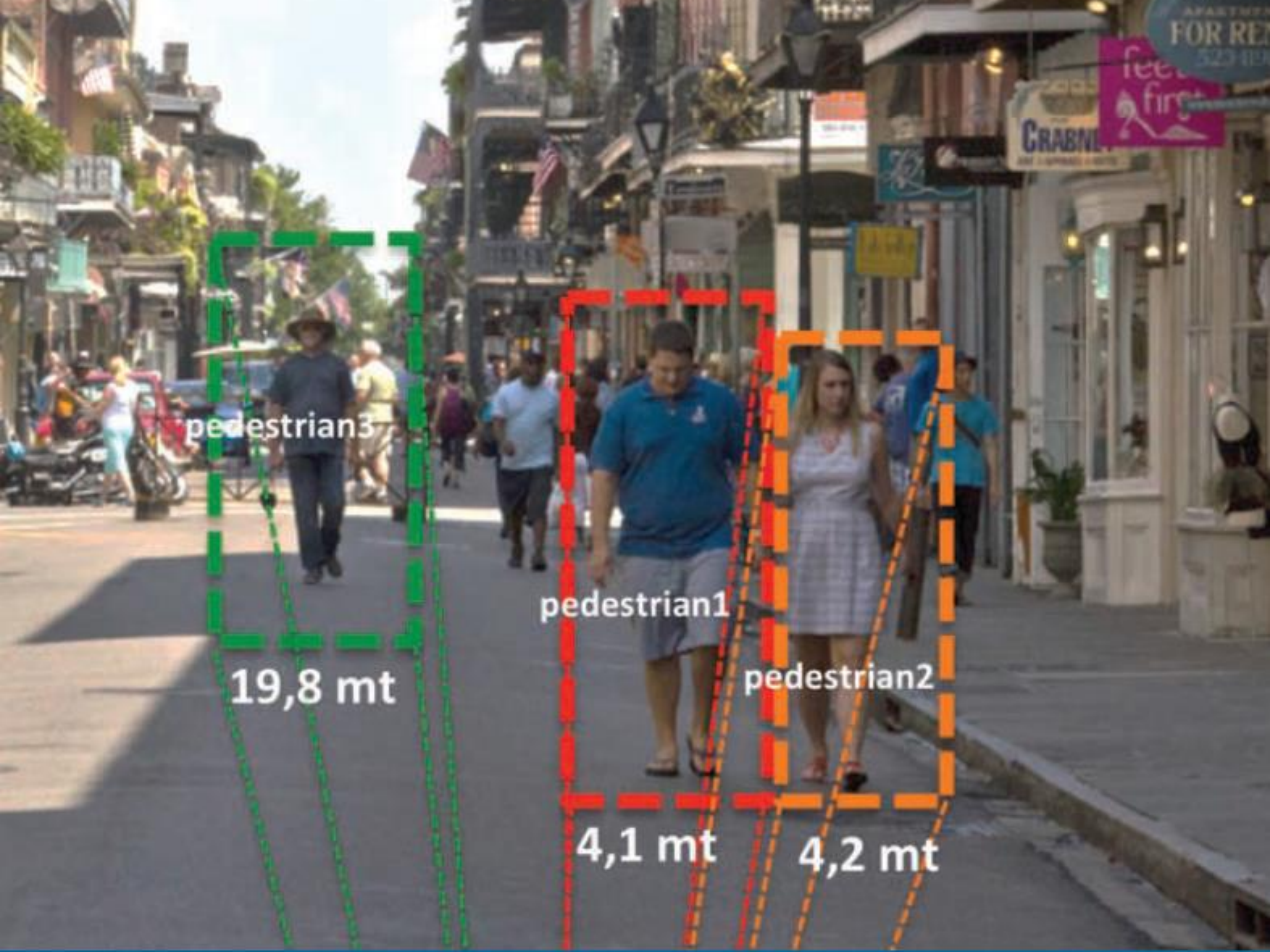
Training a DNN

In deep learning, the training itself actually functions as a programming activity. Here, for simplicity, we use the example of computer vision. To be able to match the input to the statistically expected correct result, DNN training requires a large set of image frames (typically extracted from video clips of driving scenarios) featuring shapes, edges, and colors.

58 IEEE SOFTWARE | WWW.COMPUTER.ORG/SOFTWARE | @IEEESOFTWARE

Fabio Falcini and Giuseppe Lami, Information Science and Technologies Institute of the National Research Council of Italy

Alessandra Mitidieri Costanza, Fiat Chrysler Automobiles

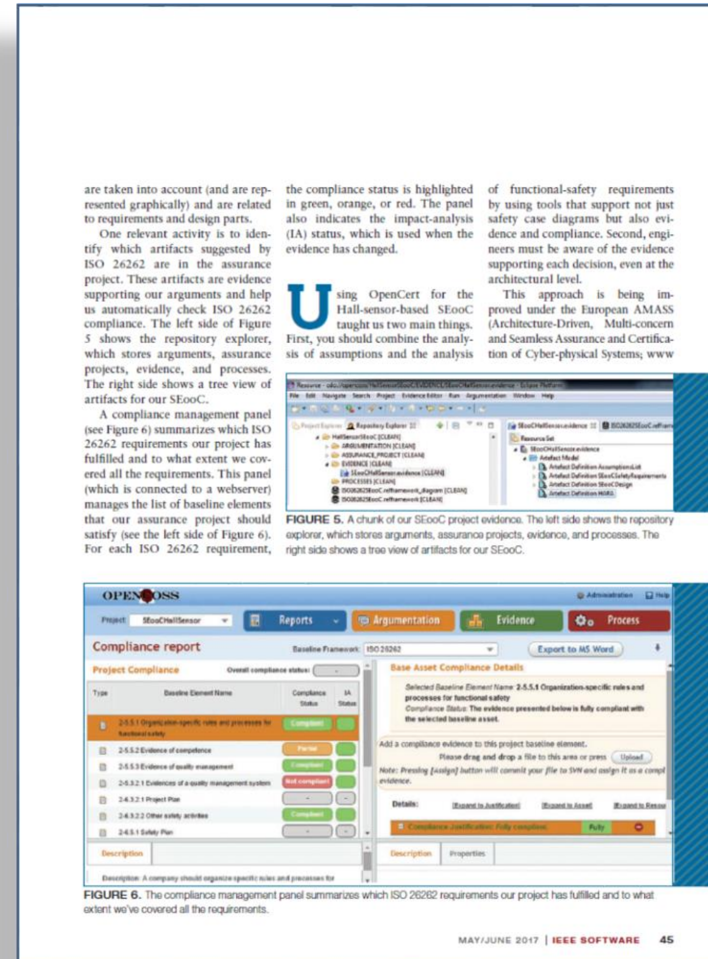

pedestrian3
19,8 mt
pedestrian1
4,1 mt
pedestrian2
4,2 mt

Supporting the Management of Reusable Automotive Software

Xabier Larrucea, Tecnalia

Alastair Walker, Lorit Consultancy

Ricardo Colomo-Palacios, Østfold University College



are taken into account (and are represented graphically) and are related to requirements and design parts.

One relevant activity is to identify which artifacts suggested by ISO 26262 are in the assurance project. These artifacts are evidence supporting our arguments and help us automatically check ISO 26262 compliance. The left side of Figure 5 shows the repository explorer, which stores arguments, assurance projects, evidence, and processes. The right side shows a tree view of artifacts for our SEooC.

A compliance management panel (see Figure 6) summarizes which ISO 26262 requirements our project has fulfilled and to what extent we covered all the requirements. This panel (which is connected to a webserver) manages the list of baseline elements that our assurance project should satisfy (see the left side of Figure 6). For each ISO 26262 requirement, the compliance status is highlighted in green, orange, or red. The panel also indicates the impact-analysis (IA) status, which is used when the evidence has changed.

Using OpenCert for the Hall-sensor-based SEooC taught us two main things. First, you should combine the analysis of assumptions and the analysis of functional-safety requirements by using tools that support not just safety case diagrams but also evidence and compliance. Second, engineers must be aware of the evidence supporting each decision, even at the architectural level.

This approach is being improved under the European AMASS (Architecture-Driven, Multi-concern and Seamless Assurance and Certification of Cyber-physical Systems; www

FIGURE 5. A chunk of our SEooC project evidence. The left side shows the repository explorer, which stores arguments, assurance projects, evidence, and processes. The right side shows a tree view of artifacts for our SEooC.

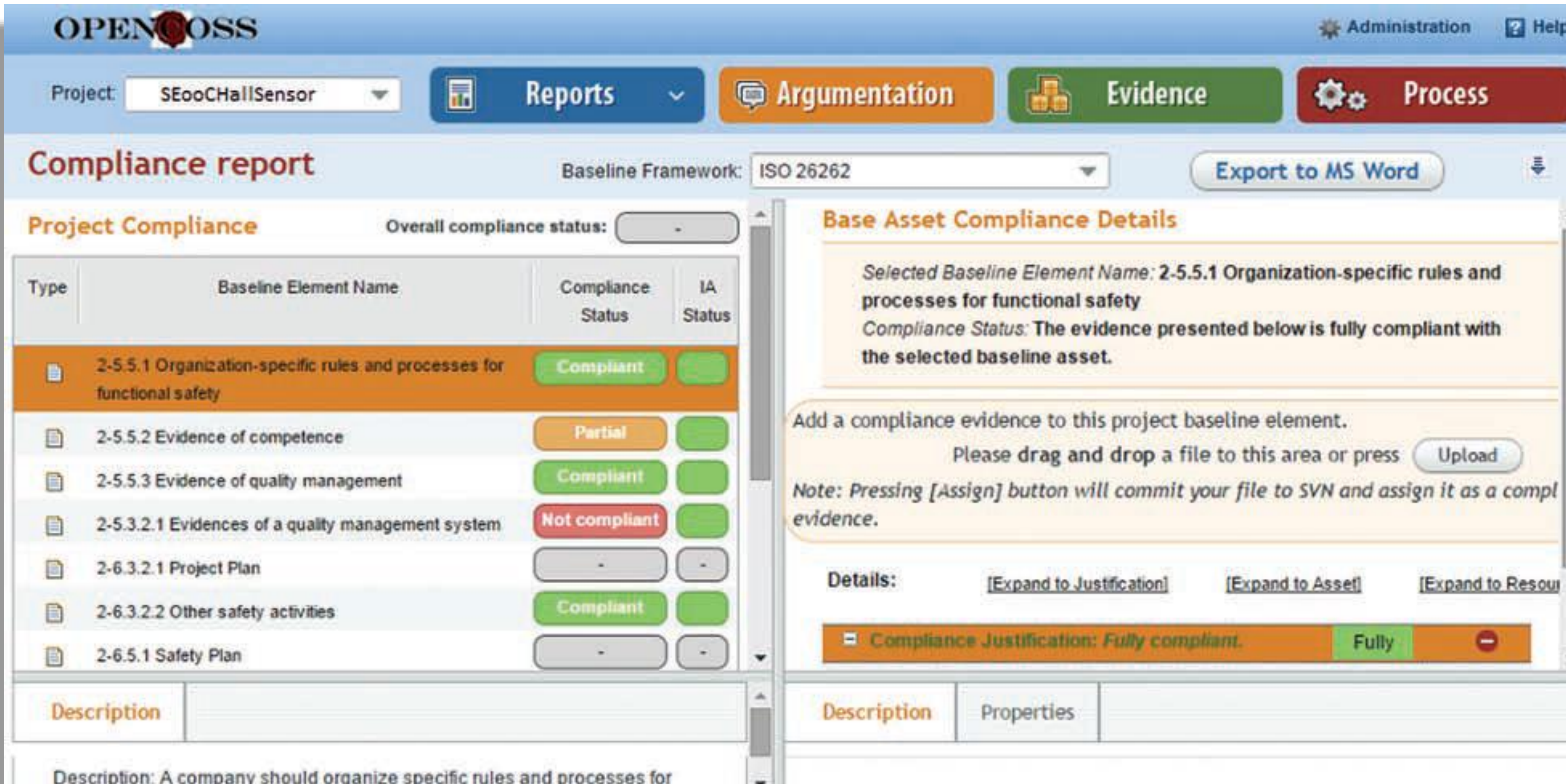
FIGURE 6. The compliance management panel summarizes which ISO 26262 requirements our project has fulfilled and to what extent we've covered all the requirements.

MAY/JUNE 2017 | IEEE SOFTWARE 45

intecs Solutions
the Brainware company

AUTOMOTIVE SPIN ITALIA

Arguing Compliance



Secure Automotive Software

The Next Steps

Lee Pike, Jamey Sharp, Mark Tullsen, Patrick C. Hickey, and James Bielman, Galois

improving software quality,[9] such as using version control, unit testing, integrated testing, and code reviews. The Motor Industry Software Reliability Association's *Development Guidelines for Vehicle Based Software* already recommends these approaches,[10] so we don't discuss them here.

The following recommendations typically go beyond the automotive industry's current standard practices. Table 1 summarizes the recommendations, organized into four areas: compile-time assurance, runtime protection, automated testing, and architectural security.

TABLE 1 Recommendations for improving automotive-software security.

Area	Recommendations
Compile-time assurance	• Static analysis • Memory-safe programming • Formal verification
Runtime protection	• System specialization • Measurement and attestation • Cryptography • Runtime verification
Automated testing	• Fuzz testing • Property-based testing
Architectural security	• Trusted interfaces • Software isolation • Glue code generation

Compile-Time Assurance

Compile-time assurance happens before code execution. We present the recommendations in the increasing order of engineering effort.

Static analysis. Static analysis tries to discover software flaws without execution or testing, and many tools are commercially available. Some tools are *sound*; that is, they shouldn't produce false negatives. To improve scalability and reduce false positives, some tools are *unsound* and can be considered advanced bug-hunting tools.

SAE J3061 recommends using static analysis, and we do too. However, although static analysis is powerful, it can lead to a false sense of security. Furthermore, static-analysis tools can produce so many false positives that discerning legitimate vulnerabilities is difficult. Finally, static analysis is unreliable for automatically uncovering domain-specific bugs.

Memory-safe programming. Microsoft discovered that the Pareto principle applies to software quality: 80 percent of Windows and Office errors and crashes came from 20 percent of the bugs.[11] We conjecture that the principle applies more generally to software security: 80 percent of exploits come from 20 percent of the classes of vulnerabilities.

For example, all the UCSD–UW attacks on short- and long-range wireless systems depended on exploiting buffer overflows.[2] Buffer overflows are a rudimentary vulnerability known since at least 1972.[12] They're a particular example of a memory-safety violation, which is an example of undefined behavior. Coding standards and static analysis target mostly the prevention and discovery, respectively, of undefined behavior resulting from using "unsafe" programming languages such as C or C++.

We propose that the most expeditious way to improve software security is to use memory-safe languages. *Safe-C* languages are memory-safe and suitable for embedded programming. They guarantee memory safety while still allowing programmers fine-grained control of memory use and timing.

We developed the safe-C language Ivory[13] to support the HACMS program. Ivory is a secure alternative to C/C++ in which memory-safety errors are impossible; it supports a variety of verification tools. The HACMS program used Ivory to develop secure avionics with no memory-safety vulnerabilities.

Formal verification. Whereas testing provides partial assurance about the actual artifact to be fielded (because one more test vector might uncover a vulnerability), formal verification provides complete assurance about a model of the system. With testing, the designer's worry is, "Have I tested enough?" With formal verification, the worry is, "Is my model's fidelity accurate enough?"[14]

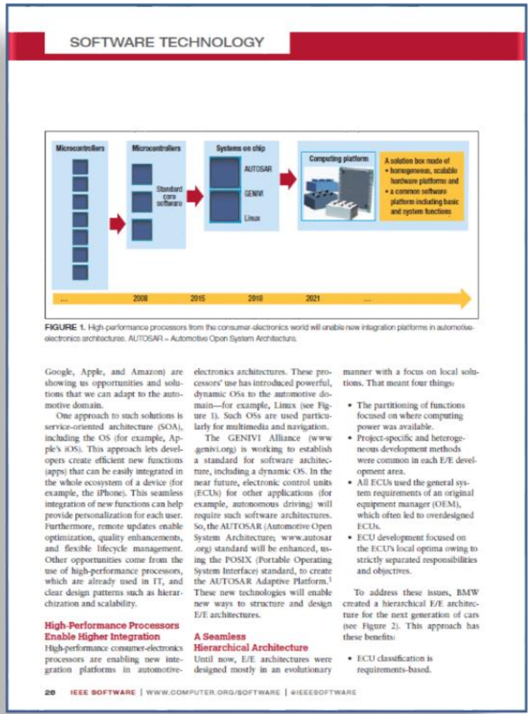
Formal verification requires a two-step approach: build a model, then verify it. Both steps are usually partly manual. Because of the effort involved, formal verification is the most cost-effective for critical, well-defined components. One example is embedded OSs.[15] Another example is specific control systems. These systems are particularly difficult to

MAY/JUNE 2017 | IEEE SOFTWARE 51

Future Automotive Architecture and the Impact of IT Trends

Matthias Traub, Alexander Maier, and Kai L. Barbehön

BMW



SOFTWARE TECHNOLOGY

FIGURE 1. High-performance processors from the consumer-electronics world will enable new integration platforms in automotive-electronics architectures. AUTOSAR = Automotive Open System Architecture.

Google, Apple, and Amazon) are showing us opportunities and solutions that we can adapt to the automotive domain.

One approach to such solutions is service-oriented architecture (SOA), including the OS (for example, Apple's iOS). This approach lets developers create efficient new functions (apps) that can be easily integrated in the whole ecosystem of a device (for example, the iPhone). This seamless integration of new functions can help provide personalization for each user. Furthermore, remote updates enable optimization, quality enhancements, and flexible lifecycle management. Other opportunities come from the use of high-performance processors, which are already used in IT, and clear design patterns such as hierarchization and scalability.

High-Performance Processors Enable Higher Integration

High-performance consumer-electronics processors are enabling new integration platforms in automotive-electronics architectures. These processors' use has introduced powerful, dynamic OSs to the automotive domain—for example, Linux (see Figure 1). Such OSs are used particularly for multimedia and navigation.

The GENIVI Alliance (www.genivi.org) is working to establish a standard for software architecture, including a dynamic OS. In the near future, electronic control units (ECUs) for other applications (for example, autonomous driving) will require such software architectures. So, the AUTOSAR (Automotive Open System Architecture; www.autosar.org) standard will be enhanced, using the POSIX (Portable Operating System Interface) standard, to create the AUTOSAR Adaptive Platform.[1] These new technologies will enable new ways to structure and design E/E architectures.

A Seamless Hierarchical Architecture

Until now, E/E architectures were designed mostly in an evolutionary manner with a focus on local solutions. That meant four things:

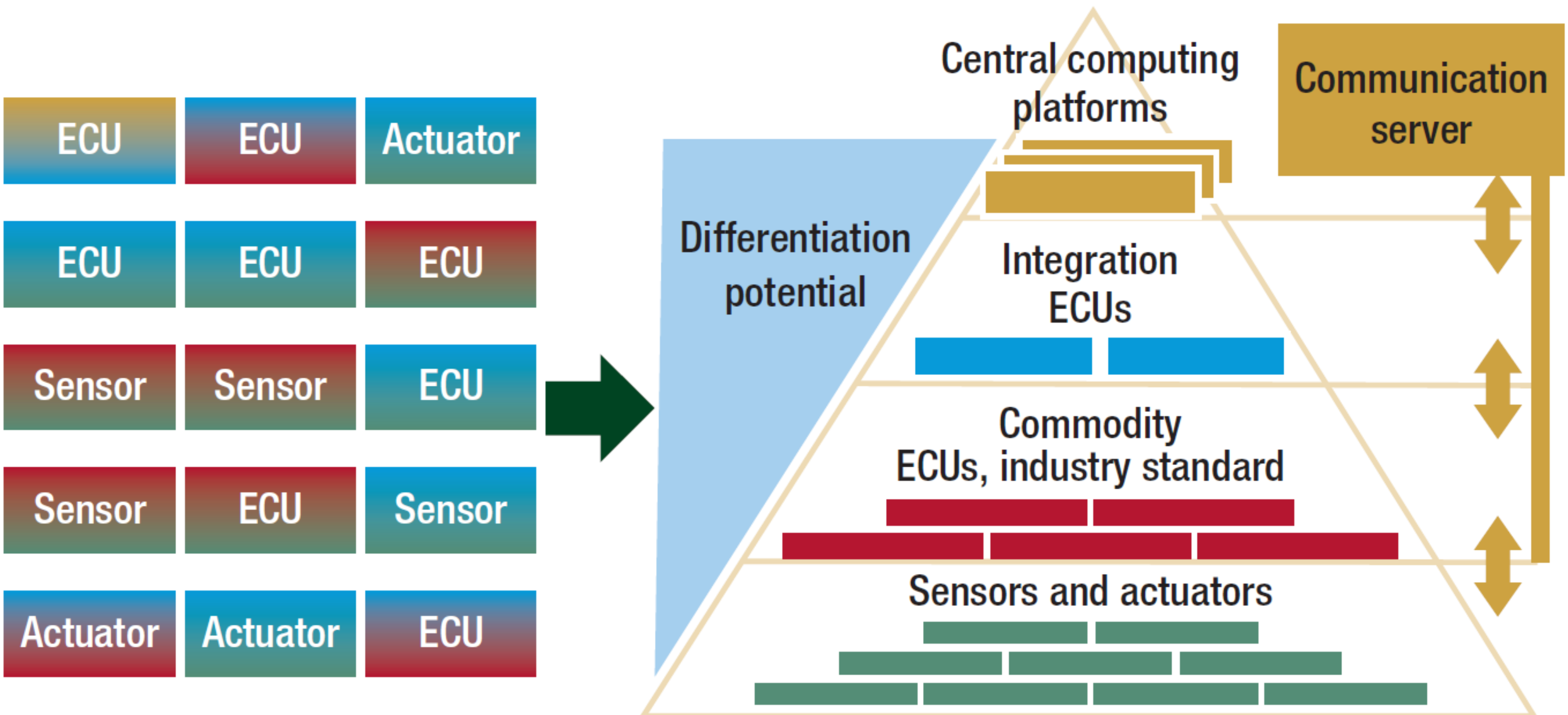
- The partitioning of functions focused on where computing power was available.
- Project-specific and heterogeneous development methods were common in each E/E development area.
- All ECUs used the general system requirements of an original equipment manager (OEM), which often led to overdesigned ECUs.
- ECU development focused on the ECU's local optima owing to strictly separated responsibilities and objectives.

To address these issues, BMW created a hierarchical E/E architecture for the next generation of cars (see Figure 2). This approach has these benefits:

- ECU classification is requirements-based.

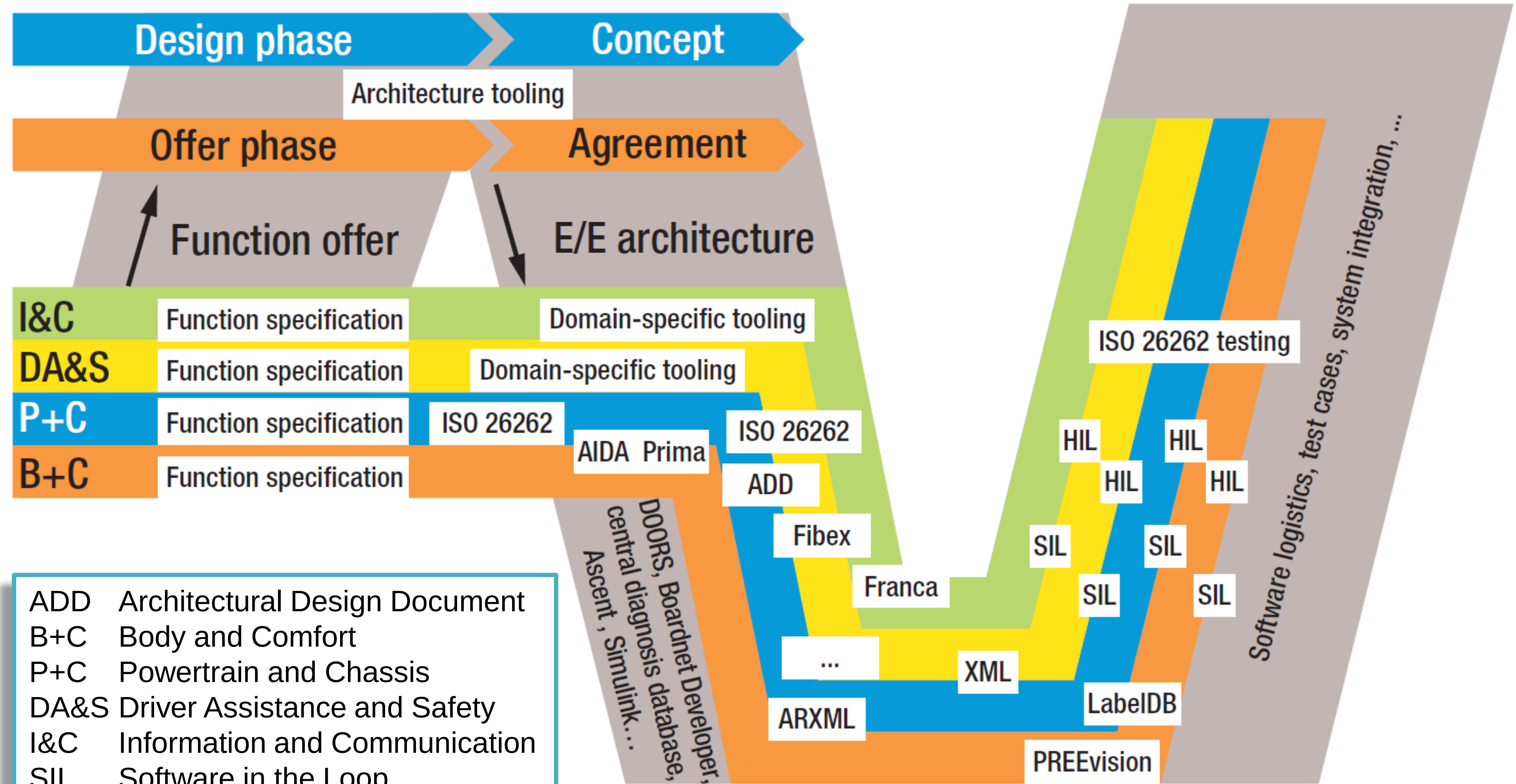
28 IEEE SOFTWARE | WWW.COMPUTER.ORG/SOFTWARE | @IEEESOFTWARE

Central Computing Platforms



(Traub et al.)

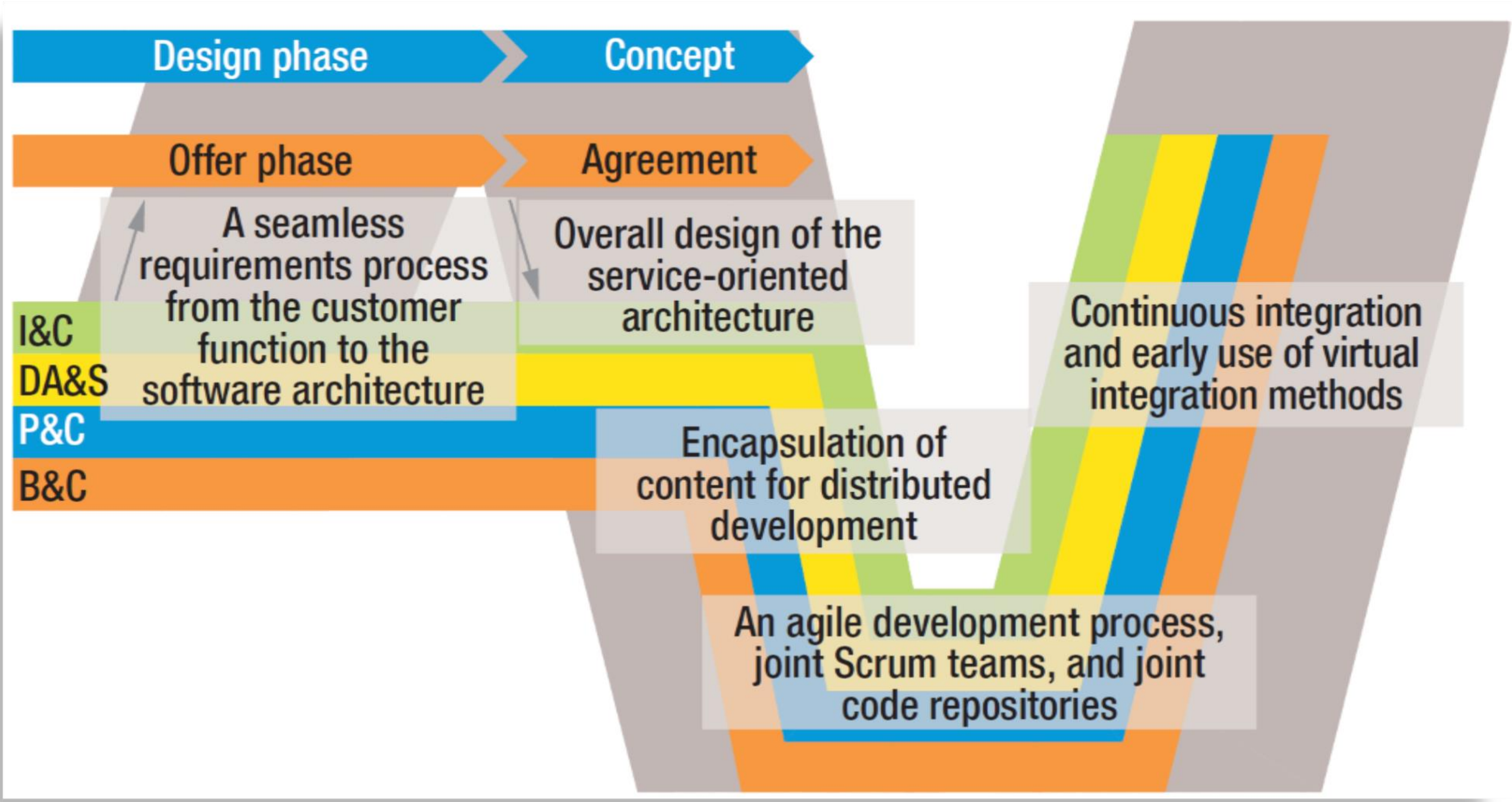
Tools – Today



ADD Architectural Design Document
B+C Body and Comfort
P+C Powertrain and Chassis
DA&S Driver Assistance and Safety
I&C Information and Communication
SIL Software in the Loop
HIL Hardware in the Loop

(Traub et al.)

Tools – Tomorrow?



(Traub et al.)

Resources

The Guest Editor Introduction to the Special Issue may be downloaded here:

`https://www.computer.org/csdl/mags/so/2017/03/mso2017030033.pdf`

The article on Future Automotive Architecture may be downloaded here:

`https://www.computer.org/csdl/mags/so/2017/03/mso2017030027.pdf`

The article on Deep Learning may be downloaded here:

`https://www.computer.org/csdl/mags/so/2017/03/mso2017030056.pdf`

There is a companion set of resources at the IEEE Software *Computing Now* site:

https://www.computer.org/web/computingnow/archive/automotive-software-may-2017-introduction

THANK YOU !

www.intecs.it