INVITED PAPER

Hardware/Software Codesign of Aerospace and Automotive Systems

Software and hardware for these systems must be co-designed since overall costs depend both on hardware requirements and on the load placed on the systems by embedded software.

By Ahmed Abdallah, Eric M. Feron, Graham Hellestrand, *Fellow IEEE*, Philip Koopman, *Senior Member IEEE*, and Marilyn Wolf, *Fellow IEEE*

ABSTRACT | Electronics systems for modern vehicles must be designed to meet stringent requirements on real-time performance, safety, power consumption, and security. Hardware/software codesign techniques allow system designers to create platforms that can both meet those requirements and evolve as components and system requirements evolve. Design methodologies have evolved that allow systems-of-systems to be built from subsystems that are themselves embedded computing systems. Software performance is a key metric in the design of these systems. A number of methods-of-methods for the analysis of worst case execution time have been developed. More recently, we have developed new methods for software performance analysis based on design of experiments. Formal methods can be used to verify system properties. Systems must be architected to maintain their integrity in the face of attacks from the Internet. All of these techniques build upon generic hardware/software codesign techniques but with significant adaptations to the technical and economic context of vehicle design.

KEYWORDS | Automotive; avionics; cyber-physical systems; design of experiments; embedded computing; hardware/software codesign; software performance; worst case execution time

I. INTRODUCTION

Modern vehicles rely on embedded computing systems for all aspects of their operation. Those computing platforms—a combination of hardware and software—are very different from the sorts of platforms used for desktop computing or server farms. High-performance embedded system platforms, like those used in vehicles, must be used to meet a variety of strict constraints.

- Real-time deadlines must be met. Safety-critical subsystems like braking and engines clearly must satisfy deadlines. However, even passenger entertainment systems must meet strict deadlines for delivery of multimedia data to avoid degrading presentation quality.
- The platform must operate using the limited power available from the vehicle's generator.
- Since weight is a concern in all platforms, the weight of the computing system and its associated power and cooling is a factor in design.
- We would like to design the vehicle as a cyber-physical system in which we match the computing system's architecture to the control tasks that must be performed.

Manuscript received March 16, 2009; revised October 21, 2009. Current version published March 31, 2010. The work of A. Abdallah and M. Wolf was supported in part by the National Science Foundation under Grants 0509463 and 0325119. The work of P. Koopman work was supported in part by General Motors through the GM-Carnegie Mellon Vehicular Information Technology Collaborative Research Lab. The work of E. Feron was supported by the National Science Foundation under grant CSR/EHS 0615025, by NASA under cooperative agreement NNX08AE37A, and by the Dutton-Ducoffe professorship at the Georgia Institute of Technology.
A. Abdallah was with Princeton University, Princeton, NJ 08540 USA. He is now with Embedded Systems Technology, San Carlos, CA 94070 USA (e-mail: aabdalla@princeton.edu).
E. M. Feron is with the Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: feron@gatech.edu).
G. Hellestrand is with Essetek, San Carlos, CA USA (e-mail: g.hellestrand@essetek.com).
P. Koopman is with the Electrical and Computer Engineering Department, Carnegie-Mellon University, Pittsburgh, PA 15213 USA (e-mail: koopman@ece.cmu.edu).
M. Wolf is with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0250 USA (e-mail: wolf@ece.gatech.edu).

Digital Object Identifier: 10.1109/JPROC.2009.2036747

Hardware/software codesign is the most widely used approach for the design of high-performance embedded computing systems. This field was originally developed in the mid-1990s to support the design of CPUs accelerated by application-specific integrated circuits (ASICs); early work emphasized topics like estimating the performance and area of ASICs to be synthesized. The design of vehicular computing platforms works with a very different set of components, but the basic motivation for codesign still remains. Hardware/software codesign have moved well beyond the design of ASICs using very small components to the design of complex networks using very complex platforms. We need to design a real-time low-power low-cost computing platform; the best way to meet all those conflicting constraints simultaneously is to codesign the hardware and software.

Vehicles present some unique requirements and boundary conditions that lead us to an updated set of codesign techniques. Vehicles generally rely on off-the-shelf hardware that is customized with software, so component selection is more important than component synthesis. The large software content of vehicles causes us to pay special attention to real-time software performance. We wish to have strict guarantees of as much of the system behavior as possible, leading us to the use of formal methods for software design in particular. The strict nature of the control laws that must be implemented pushes us to expand the codesign problem to consider control, leading us to cyber-physical system design techniques. Early codesign targets did not worry about Internet security, but today's vehicles often provide Internet access to both passengers and the vehicular systems themselves. Security must therefore become a first-class concern.

Several architectural styles have been developed for vehicular electronics. Early systems were component-oriented: each system component was a separate module, with its own electronics and usually point-to-point wiring. Federated architectures consolidate all the processors for a subsystem (navigation, for example) but do not share processors across subsystems. Integrated modular architectures share a common pool of processors among all subsystems.

Traditional hardware/software codesign [1] concentrates on what would be a single processor cluster in a vehicular electronics system. Traditional cosynthesis systems such as Vulcan [2] or Cosyma [3] synthesize a bus-based system with a CPU and an application-specific integrated circuit; codesign allocates operations to software executing on the CPU or to the ASIC. However, the design of vehicular systems requires solving much larger scale problems. Vehicular electronic systems are networked embedded systems, using as backbones a collection of diverse networks that support real-time operations. The network may have tens to hundreds of processing elements that communicate using perhaps a dozen interconnected networks. Other changes in technology since the early days of hardware/software codesign also contribute to the change in the nature of the problem.

A variety of component types can be used as nodes in the vehicular network: microcomputers and microcontrollers, consisting of CPUs, memory, and I/O devices; field-programmable gate arrays (FPGAs); or ASICs. In the past, vehicle system designers have been free to design very large-scale integration (VLSI) processing elements to support their mission requirements. But designers are increasingly constrained to using catalog components that have been designed for other applications.

The high cost of design of ASICs mitigates against their use. The design of a large ASIC typically costs \$10–\$20 million; this is a nonrecurring engineering cost before manufacturing. Given that the sales cost of such ASICs is on the order of \$10, millions of units must be manufactured to cover the cost of design. FPGAs can be used as substitutes for ASICs, eliminating the need to fabricate chips and reducing the cost of design, but FPGAs still require substantial design investments.

Vehicular system designers must also live with the fact that the lifetime of their vehicle is far longer than the lifetimes of the components built into those vehicles. Not only will the components themselves need to be replaced, but they will eventually need to be replaced with different types of parts. The current approach to this problem is for vehicular subsystem vendors to make end-of-life buys of chips as they become obsolete, and for vehicle manufacturers to purchase spare components for the lifetime of a vehicle production year in advance. This is an expensive proposition.

The economics of VLSI systems argue against part with long market lifetimes. Moore's law, which dictates that the number of transistors per chip will double every 18 months, provides constant improvements in microelectronics but also ensures that technologies are rapidly outdated. Microelectronics manufacturers drop old components after a few years because their capabilities are far outstripped by more modern parts and their flagship markets—PCs, consumer electronics, etc.—constantly design new systems. Even if manufacturers want to continue to manufacture old parts, they would face substantial challenges. The manufacturing equipment would have to be maintained for longer than the industry is used to doing; mask-making equipment would similarly have to be maintained; the computer-aided design tools used to create the designs would also have to be maintained and migrated to new desktop computing platforms in case minor changes had to be made to the components. Once Moore's law ends, system designers may be able to rely on components being available for many years. But for quite some time, system design must adapt itself to the realities of ever-changing computer technology.

Furthermore, Moore's law increasingly dictates that vehicle manufacturers must rely on commercial off-the-shelf (COTS) components and less on components specifically designed for vehicles. The design of high-end application-specific integrated circuits (ASICs) now costs

in the neighborhood of $20 million; mask costs alone can top $1 million. As these costs increase, semiconductor suppliers must seek larger markets for their parts. Many other markets do not have the strict reliability requirements or other requirements of vehicular systems. Thus, vehicular manufacturers must often make do with whatever COTS parts are available rather than use parts tailored to a particular automotive application.

Increases in computing capacity and the need to move to different processors over time dictate that more and more system functions will be performed in software. ASICs are less common in vehicles today partly due to inherent design costs and partly because engineering changes for ASICs are expensive in both time and money. Software design presents its own challenges. Some, such as version control, are similar to those of software in other domains such as business computing. Other challenges are unique to the demands of real-time and low-power embedded computing systems.

This paper surveys the design of embedded computing systems, consisting of hardware and software, for automotive and aeronautical vehicles. Even if COTS components are used, hardware/software codesign techniques must be used to ensure that the system meets the strict requirements placed on vehicular electronics. Section II describes design methodologies for vehicular systems. Section III concentrates on performance analysis of embedded software, including both worst case execution time (WCET) analysis and our new approach based on completion time and design of experiments (DOE).

II. DESIGN METHODOLOGIES

This section describes design methodologies for vehicular systems. We discuss the structural properties of the industry that influence design processes. We then overview systems engineering, introduce some terms, and discuss the major steps in the process in more detail. We will then discuss the role of formal methods in vehicle design methodologies and the challenges in making embedded computing platforms for vehicles secure.

A. How the Supply Chain Influences Methodology

Automotive system design and automobile operational environments are much more fragmented than is the case in aerospace. While we can apply some lessons from avionics to cars, many new problems must also be taken into account. The methodologies in common use in vehicle control system design are typical of many embedded development shops. Hardware dominates the architecture. But software does most of the control work, fixes the hardware limitations, and costs 70% of the engineering budget. Powertrain engineering still dominates, and Tier 1 supplier companies do most of the control system design and implementation. Suppliers largely return black-boxes to the original equipment manufacturers (OEMs, often called automakers) to assemble into cars. The engineering effort to specify and design a single electronic control system (ECU) for a car is typically based on loose, natural language specifications by the OEM or OEM and Tier 1 supplier. This odd situation in regard to specification, design, and implementation prevails largely because the OEMs largely do not possess all the IP and know-how to design and build the distributed electronic and software control systems required for their vehicles. The exception to this rule, in general, is that in powertrain engineering, many OEMs run highly advanced model specification and design processes, largely because OEMs see powertrain as both a strategic IP and marketing differentiator. Even in this case, however, Tier 1s implement the powertrain control systems and return the classic black-boxes for prototype acceptance followed by production.

Almost all non-powertrain ECUs that perform discrete control of the continuous domain physical systems constituting the controllable subsystems of a car are currently designed and realized using an objective function dominated by cost. Given the current advanced state of the electronics and software technologies, this is not a major impediment to innovation, safety, or performance. The consequence is that almost all ECUs are based on existing, tried and proven, cheap technologies and components. Essentially, the ECUs contain software executed by processors of sufficient caliber to produce computed results with required latencies and bandwidths that are communicated through buses, peripheral devices, and networks with sufficient communicational latencies and bandwidth to effect satisfactory control. There is little reason for using anything but low cost, moderate speed silicon for most control required in cars. The exceptions are high premium ECUs designed for high-end powertrain, infotainment systems, and some of the new dynamic safety subsystems effectuated by processed signals originating from external electromagnetic, thermal, and optical sensors. After two generations of silicon technology, these ECUs will themselves become low premium controllers deployed in lower cost vehicles—once again subject to the inevitable, long-term cost objective function dominated by cost.

The occurrence of Tier 1 black-box design and production in automotive engineering is not restricted to single ECUs. A number of Tier 1 companies, such as Magna,[1] Bosch,[2] and Denso[3] make substantial control subsystems containing a number of ECUs. And in Magna's case, it not only makes the control subsystems but also makes and assembles complete control systems while assembling complete cars under contract to a number of OEMs. Apart from being a strategic issue of control in the OEM–Tier 1 supply chain relationship, the balkanization that is inherent in a system architecture designed and built bottom-up using black-boxes from a number of suppliers is of concern in regard to cost, safety, reuse, ability to

[1]www.magna.com.
[2]www.bosch.com.
[3]www.globaldenso.com.

optimize, and the potential for high complexity in future control systems simply due to the restricted purview afforded any single member of the supply chain.

The concept of black-box invisibility anywhere in real-time critical control engineering, be it in automotive or aerospace systems, is prejudicial to safety and the ability to test, prove, optimize, and rationally allocate risk and liabilities in the context of the entire control system. The prevalence of black-box automotive engineering is largely a consequence of spinning out control systems and technology divisions—in the form of Tier 1 companies—by their parent automotive OEM companies a decade ago.

Belatedly, a number of OEMs have recognized their deteriorating strategic position and have drawn battle lines in an attempt to control the specification, architecture, and optimization of the control and physical systems defining their vehicles. This is likely a winning strategy accomplished by creating executable specifications together with definitive system test cases of these systems that, when translated into optimal architectures, become the means by which communication is effected throughout the supply chain. In addition, these tactics address the testing of sufficiency of design and proving that physical realizations meet their specification, again throughout the supply chain. This is the basis of the new approach to cyber-physical systems engineering that eliminates the compromises of black-box engineering and enables competition in design, design optimization, and production but not in specification, architecture, and architectural optimization.

To keep this game interesting, it is not just the OEMs who have the intellectual capacity to specify and architect the physical and control systems of vehicles. Tier 1 companies share this capability, as might a number of organizations, including research laboratories and companies—and arguably individuals. When coupled with the ability to configure short-run supply networks specified by the physical and control system architecture, the potential to disaggregate the automotive vehicle industry is present. With a global production over capacity of about 100%, the automotive industry may well be in for highly disruptive change [4]—one in which the disaggregation cycle once again rules [5]. There is also the intriguing potential for further customization of control architectures and plants that would result in extensive personalization of vehicles with the potential to conserve most of the economic advantages of long component production runs through just-in-time supply networks and to maintain the levels of regulatory compliance required to permit such vehicles to safely operate in traffic.

B. Automotive Systems Engineering and Its Methodology

The objective of the automotive engineering process is the design and production of optimized control systems that are engineered and manufactured through optimized supply chains. Both aspects of this endeavor have profound effects on the economics of vehicle control system design and manufacture, as well as traffic systems design, manufacture, and operation—where traffic is defined as many vehicles operating cooperatively and *safely*.

As shown in Fig. 1, the processes of empirical systems engineering include four key steps: the formulation of mathematical models of systems from requirements; the accurate modelling of the behavior of systems at several levels of detail; the derivation of functional models from mathematical models (mapping); and the mapping of functional models to structural models for physical

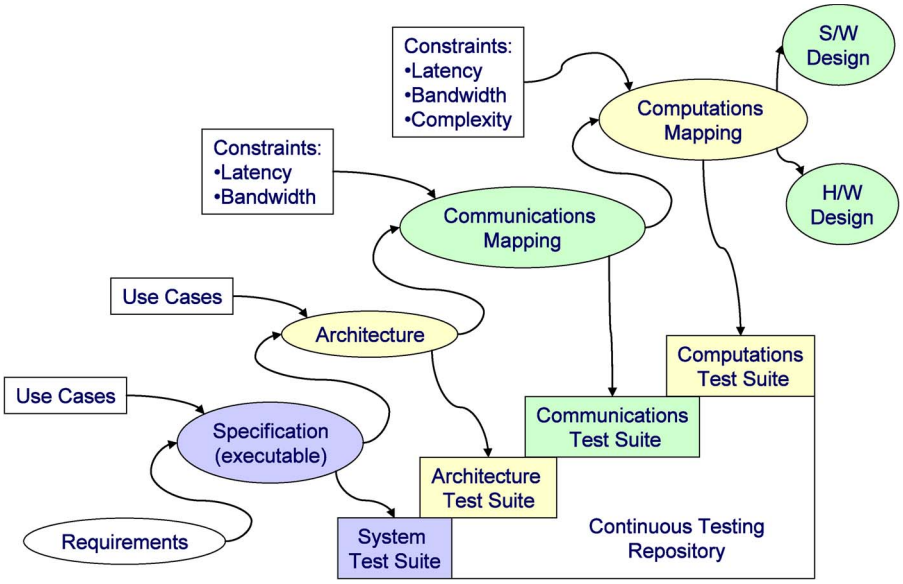


Fig. 1. ***The systems engineering process flow.***

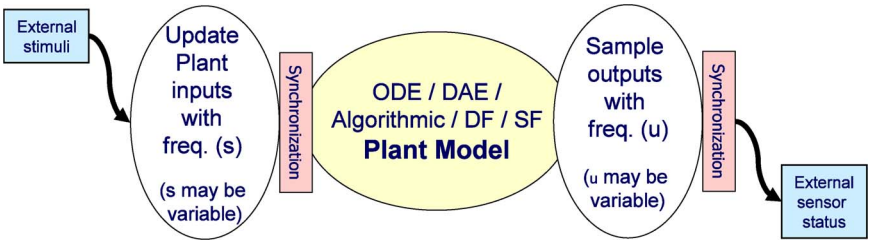


Fig. 2. ***Cyber-physical modeling.***

realization and production. Architecture is one step of a systems engineering process and lies between formal (usually mathematical) executable specification and the mapping of an architecture to software/hardware design. It happens to be a step that is highly amenable to empirical steering and optimization, as is the mapping step. The realization technologies that ground the systems engineering process, as mapping targets of structural models, are software, electronics, and mechanics (mechanical, fluid, thermal, chemical)—this is what hardware/software codesign has become in the systems engineering process. The final step in the process is the proving of the realized physical system meeting its specification. This is typically measurement based and driven by use cases both for the physical and modelling system; however, formal techniques can also be applied in conjunction with the measurement-based proving.

These tools, technologies, and methodologies, together, enable the specification of vehicle *types* that can be bound structurally, functionally, and economically to create derivatives able to be empirically optimized for various markets. This is indeed revolutionary, as it ratchets up a notch the ability to contemplate, instead of a single vehicle control architecture (VCA), a common VCA design *type* that is used to derive various VCA instantiations. These are determined by bindings that may be sensitive to the passing of time, model positioning within a family (for example, Crown, Camry, Corolla), the economic circumstances predicted for a future model release, considerations such as supply chain capabilities and component reuse, etc.

C. System Modeling and Mapping

The first step in the process, as shown in Fig. 2, is specifying the combined physical and control (cyber-physical) system to be implemented. As shown, this can be done using a number of formalisms from the preferred differential equation set (modeled and solved using packages such as Matlab[4] and Maple[5]) for continuous domain systems, to Data Flow + state machine (modeled and simulated using packages such as Lustre [6], Simulink), to functional programming/data flow (Functional Programming [7], Modal [8], Lisp [9]) for discrete domain systems. Ideally, the formalism chosen is mathematically well connected to support reasoning about the system. Synchronization is required for input and output with internal states and function computations.

The second step, as shown in Fig. 3, is to separate the control specification from the physical system specification. This is not a trivial step and requires decisions to be made about what constitutes the physical system (called the *plant*) and what constitutes control to be applied to the plant, either as part of a feedback loop or as external input information. Synchronization is now particularly important to ensure i) data from a combination of feedback from the control model and input will coherently drive the actuators and ii) the capturing of stable state and output information from the plant model to be used in computing the updates of the controller, which is to be applied at the next synchronization event to the plant model.

In summary, from a *parent* specification undifferentiated with regard to a physical systems and its control, it is possible to describe—perhaps even generate—a potentially large set of functionally equivalent specifications in which the control is separated from the physical system. Typically, the physical system has a continuous behavior in time. The control part of the separated system (called the controller) may also exhibit continuous behavior. When reduced to engineering practice, almost invariably the controller is discretized with respect to time and data values, and the control behavior may be realized as either software executing within the ECU or, on the rare occasion when it is warranted in automotive systems, as hardware. This is the standard codesign deployment, and a number of companies (for example, The MathWorks, D-Space, and Esterel Technologies) facilitate the efficient conversion of abstract controller specifications to hardware or software realizations.

D. Systems Engineering Process: Specification → Realization

There is more than one way to separate control from plant, in reality and in modelling. A surprisingly common approach to solving this problem is to build a physical prototype of the specified control system, measure it, and

[4]www.mathworks.com.
[5]www.maplesoft.com.

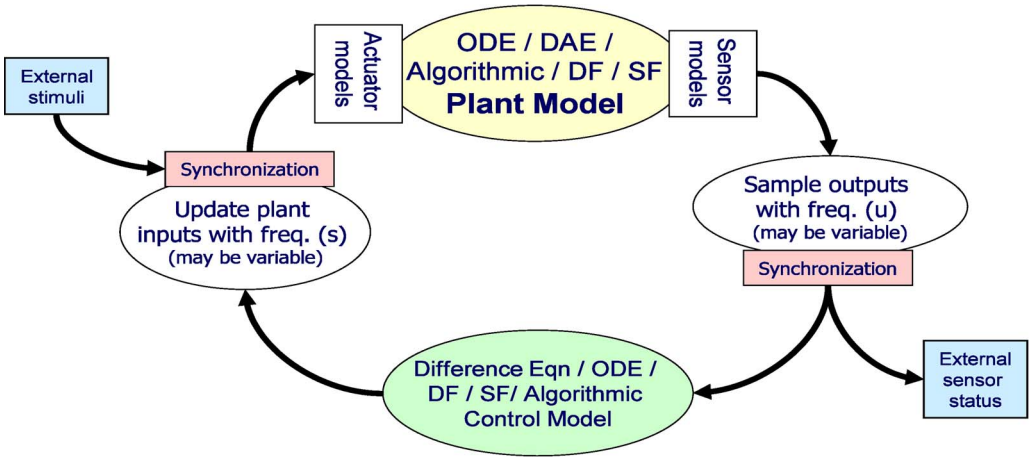


Fig. 3. ***Separation of control and physical (plant).***

then use the measures to tweak the prototype and declare it a production control system architecture. It is no surprise that no one has any idea how far from optimal the production control system might be. And it is ironic that poor methodology in basic engineering is likely to compound economic damage through the production cycle of each new model in which the VCA is reused.

The optimization of the VCA is of paramount importance. Fig. 4 shows the two of the four stages in the systems engineering process where it is directly applicable.

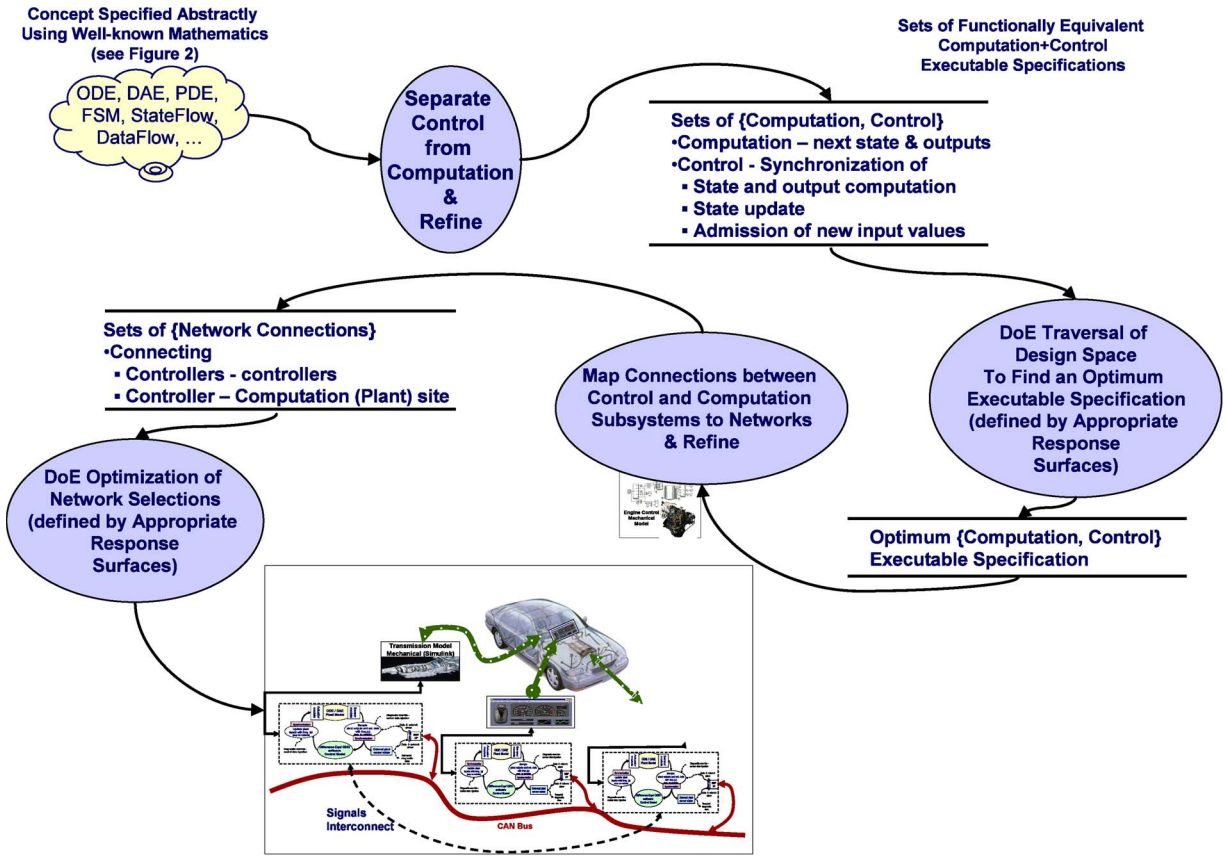


Fig. 4. ***The systems engineering process from specification to partial realization.***

The *Mathematical Separation of Control from Computation (Plant)* phase of the systems engineering process in Fig. 4 shows the proliferation of control system candidates. That each candidate is a simulatable model is fundamentally important in the creation of an efficient engineering process, enabling the dynamic characteristics of each candidate control system to be measured and used in the optimization process. The existence of models enables the optimization process to essentially walk through the design space, and use the simulation results from each, to determine those that belong to the set designated as optimal according to some objective function. At this stage, the controller models are still reasonably abstract—but each has the potential to have both hardware and software realizations, with computable complexity, and is associated with a set of plant models of known automotive function having measurable information latency, bandwidth, and frequency requirements. This is captured in by the *DoE Traversal to Find the Optimum Executable Specification* process in Fig. 4.

The *Mapping Connections between Control and Computation (Plant) to Networks* process is entered with a small number of candidate VCA controller architectures. The mapping of nets between controller and plant models to physical networks and net models, with known attributes, is relatively straightforward and is not further discussed here. The selection of which network mapping is optimal (process 4 in Fig. 4) is amenable to efficient DoE design space traversal and optimization.

Of major importance for the distributed control system of a vehicle (automotive, aircraft, and naval) is its architecture, typically, a VCA of hierarchical and distributed subsystems (for example, in a car—stability control, powertrain, braking, cruise control, chassis, steering, body, etc.). Structural factors determine how distributed or centralized an optimal control system (or a family of control systems) will be. The control functions associated with a particular physical plant (such as engine, transmission, individual brakes, etc.) are typically carried out by some ECU. Where the ECU is located and which tasks share the ECU are determined by other factors that optimize the overall architecture and determine the network (intercommunication) structures between, and the task scheduling structures within, the control system ECUs.

The final phase in Fig. 1, *Computations Mapping*, is the mapping of controller models to hardware or software; + ECU Realizations is a classic codesign problem for each controller. Once again, DoE can be used to efficiently choose an optimal realization across controllers for the whole VCA. We have seen a DoE protocol described above, and it is assumed that codesign techniques are known to the reader, so these will not be repeated here. What is of interest, however, is that it may be possible to map multiple controller functions (actually more likely to be threads or processes) to single ECUs as one of the realizations of the VCA. To do this, we will need to demonstrate that for a given ECU, all of the control threads are able to guarantee that they will meet their specified targets. This requirement can be formulated as a set of classical real-time scheduling problems that, apart from relatively trivial cases, requires extensive simulation to help solve. Section IV discusses this and demonstrates—perhaps unexpectedly—the use of DoE in determining the worst case execution time for threads operating on a single ECU. The WCET problem for an interconnected, real time, highly distributed control system is more complex, and its discussion remains for a further paper.

Vehicle architecture can be optimized according to objective functions that have aspects of behavior, timing (latency), structure (e.g., fault tolerance, location, repetition), cost (e.g. of engineering, production, assembly), performance, etc. Optimizing the vehicle without accounting for the context in which they operate virtually guarantees wildly suboptimal outcomes. Two examples of the current suboptimal approach to automotive vehicle design are essentially calibrated, that is, have their control parameters set, once in their life: 1) if engines are turned off while vehicles are stationary—say, at stop lights, fuel consumption is decreased by about 17% and emissions are reduced by a comparable amount—a trivial optimization step with a big impact; 2) currently the real-time control system of automotive vehicles, which possess several hundred tunable parameters. This calibration is set so that a vehicle model exhibits *expected* performance—dictated by the marketing folk—across all terrains and operating conditions. On some vehicles, this scheme is sometimes dynamically modified to account for wear of vehicle components. Overall, this approach means that a whole vehicle model—maybe hundreds of thousands of individual vehicles—use the same engine, braking, chassis, etc., parameter settings whether they travelling on a flat road, mountainous road, in sunshine, rain, or snow, in traffic or in splendid isolation. While this approach makes mass production easier, it is suboptimal when there is technology to enable a vehicle to be continuously calibrated depending of terrain, traffic, component age, infrastructure constraints, inter alia. The effects on fuel consumption and emission control can only be guessed since there are no data available, but reductions much bigger than the 17% reductions cited for turning the engine off while stationary are highly probable.

E. Formal Methods for Software Design

Software design and implementation in automotive and aerospace systems must be especially considerate of safety concerns. Indeed, any misbehavior involving these vehicles and devices can be life-threatening, and the failure of several aerospace systems ultimately has been blamed on software errors or closely involving software functions. Some memorable incidents include navigation software malfunctions on the U.S. air superiority fighter F-22 when crossing the International Date Line, and the European

Ariane V rocket inaugural launch failure due to corrupted inertial measurement data. A good account of the interrelation between system safety and computers can be found in [10]. System failures eventually attributed to software errors are the target of numerous criticisms because they are concerned with what could be described as the most *man-made* component of the system. Other system components (e.g., structural components), although thoroughly *designed*, can always be still subject to yet-unknown failure modes, which renders the boundary between man-made errors and acts of God blurrier. This is the case, for example, with early structural failures of Boeing's 787 Dreamliner. In the case of software, it can be argued that such a boundary does not exist and that all accidental software failures could have been prevented with enough human input. The outcome of these concerns has been a progressive awakening of the embedded software community to the need to exercise thorough quality control on the software running on large airborne platforms. Software quality control may be exercised through process control, on the one hand, and product control, on the other hand.

Process control consists of making sure that software development, analysis, and monitoring follow published guidelines during and after software development and implementation. Such process control guidelines are outlined in documents such as RTCA's DO178B [11]. A close inspection of software development processes may indeed bring several deficiencies and vulnerabilities to light, even without looking at the software itself [12]. In general, the processes outlined in [11] are independent of available software verification technologies and insist on "good engineering system development practices" while leaving the detailed implementation of such practices up to individual developers, according to the state of advancement of software analysis technology.

Product control consists of a set of techniques aimed at directly evaluating software conformance to its specifications. We refer the reader to [13] for a detailed and rigorous presentation of these techniques.

The simplest software control technology is testing, whereby the software of interest is run on a variety of inputs of interest at different stages of its implementation, and its response is evaluated against a set of performance requirements. Beyond obvious expected functionalities, other requirements have been designed as attempts to address structural questions. These include, for example, code coverage requirements (all lines of code must be effectively executed) and variable coverage requirements (the test suite must make sure that each variable, taken individually, reaches all possible values). The goal of this time-consuming task is to isolate the conditions under which the software misbehaves and to fix it appropriately. Coverage requirements thus form very poor and incomplete forms of state-space reachability analyses. Software may be tested at various levels of implementation, beginning with the specification level, where both software and hardware components are represented as executable computer programs ("software simulation") and ending with the implementation level, where in the latter case many of the system's hardware components are included in the simulation so as to emulate the system as close as possible to actual operational conditions ("hardware-in-the loop simulation").

Testing technologies, although they are a necessary and very useful element of the software debugging panoply, are incapable of providing more information than that collected from the finite set of conditions under which the software was simulated. The role of static analysis methods is to complement tests with software and software specification evaluations that offer possibly conservative, yet comprehensive coverage of all possible software behaviors, and therefore global conclusions about its performance. Most static analysis methods will generate false error alerts; however, they will miss no error. Methods for the static analysis of safety-critical software (e.g., formal methods) vary depending on the level of implementation of the software under consideration.

At the specification level, many of the available analysis techniques are those available for the analysis of continuous or discrete dynamical systems and can be found in standard textbooks on automatic control [14]. Such analysis techniques aim in general at establishing stability and robustness properties of the system specifications, using well-established measures, such as gain and phase margins and Lyapunov stability theory. The development and extension of these analysis methods constitute the core of the research in automatic control systems, with particular emphasis given to the analysis of nonlinear and adaptive systems. Methods aimed at analyzing discrete and logical dynamical systems include such analysis tools as SPIN [15], [16], a model checker, and PVS [17], [18], a theorem proof assistant. Unlike many of the criteria and metrics available for continuous systems, the verification environments provided by model checkers are computationally very intensive because the structure of the underlying state spaces is usually much more complex than those tackled by continuous automatic control systems.

At the level of code analysis, much progress has been made since the early foundations of the discipline were laid down more than 40 years ago [19], [20]. The recent developments include i) a better understanding of how to mechanically express the code semantics to establish the properties sought or inquired about and ii) algorithmic environments powerful enough to establish the properties sought within acceptable deadlines and without excessive conservatism. These developments include adaptations of model checkers that take source code as input to automatically create the models to be checked. They also include tools such as ASTREE [21], which is able to automatically analyze complete avionics codes for aircraft within a few hours and prove the absence of run-time errors, for example. One of the key aspects of these recent

tools is the development of abstraction mechanisms, whereby complex system behaviors are conservatively approximated by simpler objects when performing the reachability analyses necessary to identify the possible presence of run-time errors. By a strange twist of luck, such tools make extensive use of concepts also known by the dynamical systems community, building an immediate bridge between such code-level analyses and common specification-level analyses developed by dynamical system specialists [22]. At the same time, some of these techniques are not readily usable for *system* analysis, and it has been recognized by many that, in that regard at least, it is necessary to replace hardware systems by *models*.

Such a view, which has dominated the model-checking community, is now making progress elsewhere as a necessary condition for analysis methods to bridge the gap separating *software* analysis from *system* analysis.

To ensure the future of system analysis methods, several directions must be taken. The first direction consists of the ability to develop the necessary proofs of system behavior. With a few exceptions, detailed thereafter, proofs are easily developed earlier rather than later in the software development process. Indeed, the compactness and domain-specificity of notations at the specification level make it convenient to develop convincing proofs. One exception to this rule is when the object of the proof escapes the requirements expressed at the specification level. This is the case, for example, of WCET, discussed thereafter, and of behaviors induced by the use of fixed-point or floating-point arithmetic, whereas specification-level analyses usually consider real numbers instead.

A second direction is the ability to migrate proofs from highly abstracted requirements down to the most concrete code implementations—either directly or indirectly. Indirect proof flow-down is directly concerned with the process whereby the various levels of code implementation are guaranteed to perform as expected, using a combination of proofs that i) the specification that originate the implementation is proved to be correct and ii) the implementation is a faithful simulation of the specification. Such an approach is attractive in the sense that the "burden of proof" is carried only at the specification level, and the only ensuing proofs are that the successive layers of implementation are appropriate simulations of each other. This approach essentially forms the core element of "credible compilation" as proposed by Rinard [23]. Direct proof flow-down is concerned with translating the proof(s) supporting the good behavior of the software from requirements to more and more concrete software implementations. The core idea behind this task is that the proof should be closely tied to the given software implementation. Thus, proof translation may mean much more than a simple transcription from a given high-level to a lower level. Such proof translation may also face hidden issues, for example, the models used at the highest level of abstraction do not capture exactly all aspects of its eventual implementation. It then becomes important to weigh the relative merits of i) "moving the modeling issues upwards," thereby leading to more complex, but more exact abstract specification models or ii) "adapt the proofs downwards," that is, adapt the proofs (or check that the existing proofs work) to the more concrete level of implementation. One of the possible advantages of direct proof flow-down is that since the proof is accessible for all levels of implementation, there is no immediate need to access ALL levels of system implementation to verify the proof of proper execution at a given level of implementation.

F. Security

Aircraft and auto control systems are increasingly being connected to external computing systems. There are truly compelling market reasons to do so, but this connectivity raises significant and challenging security concerns. A number of different architectural approaches to connecting control systems to desktop and enterprise systems have been proposed (representative approaches are described by Wargo [73] for aircraft and Polishuk [74] for automobiles). Approaches can be expected to vary, but both aircraft and autos will have remarkably similar issues to deal with in that they involve interconnecting many different types of components with different roles to play in terms of criticality, data types, real-time performance, and security concerns.

Fig. 5 gives an example of a generic vehicle information architecture that largely applies to both aircraft and autos (the term "vehicle" in the figure is intended to represent either type of vehicle). There are two main points to the figure. The first is that different types of computing

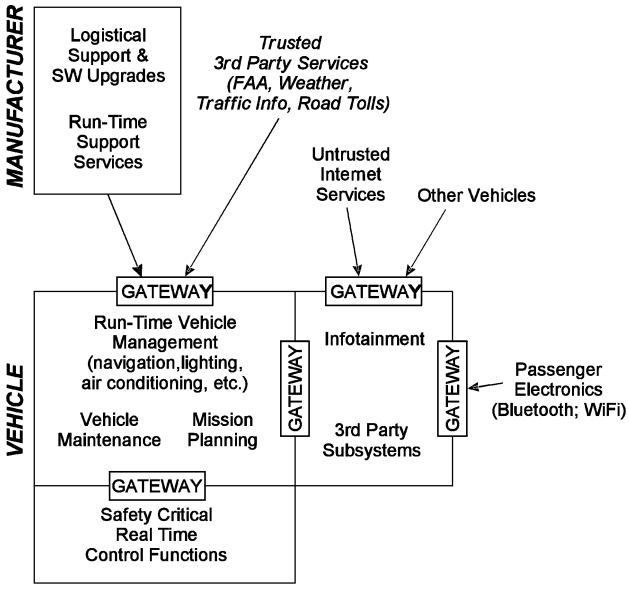


Fig. 5. ***Security aspects of vehicle information architecture.***

functions are separated by gateways of some sort. The second is that there are many different types of functions that all interconnect directly or indirectly, with varying levels of trustworthiness. In particular, almost any computer in the world can connect to safety critical vehicle control functions if it can get past two or three gateways.

It is futile to attempt to isolate safety critical control systems from other systems in the network. While it is clearly undesirable to permit an untrusted Internet source to operate flight controls on an aircraft, it is highly desirable to support operations such as flight control interaction with mission planning and maintenance software; mission planning software feeding information to the cabin infotainment software to display estimated time of arrival; infotainment system interacting with passenger laptops to support Internet access; infotainment system interacting with the Internet; and vehicle maintenance functions interacting with manufacturer information systems (for example, to support advanced fault diagnosis). So the problem is not that anyone would try to make it possible for someone on the other side of the world to operate an aircraft's jet engines or an automobile's steering in real time. The problem is that the transitive closure of connectivity for obviously desirable functions means that, in the end, everything is indirectly connected to everything else.

Thus, getting the gateway right is critical. Unfortunately, it is difficult to get enterprise gateways (or firewalls) perfect. And we know even less about what needs to go into a gateway that connects embedded to enterprise systems.

Understanding what goes in an embedded system gateway of this type is an open research question [75]. But, understanding the constraints, assumption differences from enterprise firewalls, and necessary capabilities of such a gateway is a starting point. In particular, gateways tend to join portions of a system that have dramatically different design tradeoffs, and it is often unclear what to do when those tradeoffs clash within gateway operations. Some of the more important differences driving those tradeoffs are as follows.

Control Versus Events: Most enterprise computing deals with transactions or events. Something happens, the result is committed to a database or sent to an output device, and then the next thing happens. But much of embedded computing deals with control loops that must repeatedly adjust output values based on not only input values but also the history of those input values. Thus, embedded systems tend to be time-triggered (based on periodic real-time computation rather than discrete events). These differences may require significant design consideration in transforming data and timing across gateway interfaces. They may also make systems vulnerable to timing-based attacks across gateways. These differences may require significant design consideration in transforming data and timing across gateway interfaces ([76] is an initial step on understanding these issues).

Native Support for Security: Enterprise computing systems have varying levels of support for security built in or available as standard add-ons. Not so for most embedded systems. Due to cost, size, and a history of security based on physical isolation, most embedded systems have essentially no security functions. For example, predominant automotive and aviation embedded control networks have no support for authentication or encryption, and at most primitive protection for denial of service attacks. In a typical embedded system, if an attacker can gain control of a gateway, he/she can issue any command they want on the network without much effort. Worse, embedded networks generally have small message sizes and scarce bandwidth, so overlaying enterprise security mechanisms on them is unlikely to work [77]. Secure microkernels (e.g., Integrity-178B[6] and OKL4[7]) may assist with distributed embedded security. However, many controllers within automotive embedded systems are small processors that do not run a real-time operating system as such.

Need for Stability Versus Need for Security Patches: A lynchpin of enterprise security is periodic patches of vulnerabilities as they are discovered. Over time, a race has developed between attackers who discover new vulnerabilities and patchers to attempt to fix and distributed software updates before too many systems are compromised. But, in the embedded world, a high premium is placed on software stability, and periodic (weekly or even daily) software patches are almost unthinkable. In part, this is due to the issue of recertifying a system as safe (or at least adequately dependable) after a software patch. In part, this is due to the typical lack of a system administrator to fix inevitable problems when automatic patching does not work quite as planned. While we may see frequent patches to infotainment-type software in vehicles, it is not so clear how to make releasing security patches to safety-critical vehicle control functions inexpensive or quick enough to be effective.

Automotive and aircraft embedded system security have many similarities, but there are also some important differences. The source of many difference is that aircraft operation and maintenance is highly controlled and regulated, whereas cars are far less regulated, making automotive security a potentially more difficult problem to solve.

Certified aircraft mechanics are trusted to avoid knowingly compromising security. But it is common for everyday car owners to modify their vehicles in unapproved ways, even to the point of subverting safety mechanisms in critical software. For example, unauthorized software modifications are commonly made to gain engine performance. Thus, automotive security may need to address increased threat levels from the vehicle owner and maintenance technicians that are not present in aircraft.

The design and security approach for an aircraft is likely to be more thorough and more defect-free than

[6]http://www.ghs.com/.
[7]http://www.ok-labs.com/.

approaches for typical automotive systems. This is because aircraft go through rigorous independent certification processes and lifetime configuration management. In contrast, while automobiles are remarkably reliable and safe given their complexity and extreme cost constraints, they simply are not scrutinized as closely as aircraft. Thus, it is only reasonable to expect that they will be deployed with more potential security issues.

Overall, we can expect security issues to increase in visibility and importance for both aircraft and automobiles over time. Interconnectivity and vulnerabilities are already there, and will surely increase in potential for security problems over time. Fortunately, vulnerabilities due to the issues inherent in creating embedded to enterprise gateways have not (yet) been exploited by attackers in any significant way.

Security Architecture and Methodology: Because automotive and aerospace systems are usually created as a set of black boxes, the security architecture has to be isomorphic to this set of black boxes and their interfaces. (In other words, we have to live with the set of black boxes defined for functionality and figure out how to add security to it while keeping each piece of security within a predefined black box.) As a practical matter, most architectures have network gateways between layers of the control and system hierarchy (for example, between the navigation layer and the vehicle control layer). The usual approach to security is to add security functions into these existing gateways to isolate one part of the system from attacks coming from another part of the system and to isolate the system as a whole from external attacks. Techniques for building security into systems built of black boxes remain an open research problem.

III. SOFTWARE PERFORMANCE ANALYSIS

Real-time software requires careful attention to software performance. We start this section with a survey of research on WCET analysis, which has received a great deal of attention over the past 20 years. WCET analysis finds tight bounds on the absolute worst case execution of software executing on a particular processor, independent of any input value dependencies. WCET analysis is well matched to deadline-oriented software design. We then introduce new methods for software performance analysis using the well-known DOE methodology. This approach develops a statistical model of software execution time and identifies the factors that determine the observed variations. This approach is designed for design methodologies in which deadlines may be flexible.

A. Worst Case Execution Time Analysis

Wilhelm *et al.* [78] provided a comprehensive overview of the topic of worst case execution time analysis. In this section, we briefly survey some techniques proposed to solve the WCET problem.

WCET analysis can be broken into the following phases [24]:

- path analysis or high-level analysis inspects the program's source to extract feasible flow paths;
- transformations translate the flow information obtained in the path analysis, along with the control structures from the high-level representative language to an intermediate form for WCET analysis;
- target modeling estimates the execution time of individual instructions from the target hardware;
- WCET estimation determines the actual worst case based upon the facts derived in the earlier phases.

Path analysis takes on the crucial role of examining the source code of the program under consideration for WCET analysis in order to obtain all paths that can theoretically be taken by this program. Path analysis extracts flow facts that provide details regarding the constraints on some of the potential paths: bounds on the number of loop iterations, the nesting of if-statements, etc. Because the number of paths grows exponentially, certain approximations need to be made. Any approximations must be conservative to avoid violating the "safety" condition that all viable paths are included in the results.

In an effort to more effectively prune out the nonexecutable paths from the flow path set, some have looked to symbolic execution for the answer. The idea as implemented by Chapman *et al.* [25], Altenbernd [26], and Stappert and Altenbernd [27] runs on the control flow graph of the program. Lundqvist and Stenstrom [28] also perform symbolic execution, but at the instruction level rather than abstract structures. In a variation on that approach, Ermedahl and Gustafsson [29] use abstract execution, which executes the program abstractly, by keeping abstract values for the variables, such as intervals, and performs abstract operations on them. They guarantee that an abstract value interval represents a set that always contains all possible concrete values, so they can use this to remove false paths. Ferdinand *et al.* [30] also perform abstract execution to remove false paths, but they do so at the object code level for processor registers rather than program variables. Healy *et al.* [31]–[33] implement algorithms to automatically determine iteration bounds for loops with multiple exits and to automatically obtain averages of iteration bounds for loops with conditions depending on the values of counters of outer loops. Colin and Puaut [34] also propose to count loop iterations of nested loops using annotations that store expressions of maximum iterations dependant on outer loops rather than constants. Another approach was taken by Holsti *et al.* [35] to bound loop iteration counts, but they limited it to loops that always increment a counter with a constant value. Relying on the heavy correlation between input data and control flow, Ziegenbein *et al.* [36] use these data to

automatically remove false paths. Wolf and Ernst [37] also look to take advantage of this relationship by creating a syntax graph composed of control structures as nodes and successor edges.

In practice, WCET tools make use of user-provided constraints such as loop bounds or input value ranges to improve the quality of their result. Flow facts need to be mapped from the high-level language source to the object code in order to properly determine the timing of those paths. For the goal of using automated support in completing the transformation, Engblom *et al.* [38] implemented the cotransformer. Their tool works in conjunction with the transformation trace emitted by the compiler to describe the transformations in their own optimization description language. These are then positioned in the code by matching the function and basic block names from the flow graph nodes to the trace. Kirschner and Puschner's approach [39] is to modify the open-source GCC compiler to produce assembly code annotated with timing. As opposed to the previously described approach, this only works for a single language and compiler combination. Another technique in avoiding a direct translation stage was implemented by Healy and Whalley [40]. They instrumented the compiler itself to detect effect-based and iteration-based constraints, and to generate the relevant path information using those constraints.

The most active field of WCET analysis is the one involved with researching the topic of execution modeling. For instance, Lim *et al.* [41] keep a timing abstraction data structure representing tables that contain timing information of an execution path that might be the worst case. Kim *et al.* [42] extend this approach to better improve this dynamic load/store data caching behavior by assuming two misses for each reference to one of these. They then implement a second pass to reduce this miss overestimation by identifying these dynamic references in loops and removing extraneous identified misses. In either case, there is overestimation of the cache misses. Li *et al.* [43] come up with a different data structure, the cache conflict graph, for each cache set containing two or more conflicting blocks of contiguous sequences of instructions. The data cache uses a variation of the cache conflict graph focusing on load/store instructions. Lundqvist and Stenstrom [44] implement a cycle-level symbolic execution in order to model the timing of processors with separate instruction and data caching. Ottosson and Sjodin [45] assume that each memory access is a miss. Ferdinand *et al.* [46] employ a different approach by separating the cache profiling from the calculation stage. This separate profiler implements abstract cache states that keep upper bounds of ages of memory blocks to determine if a memory block is definitely in the cache. Mueller [47] implemented a separate static cache simulator, which works only on the instruction cache. White *et al.* [48] also use a separate static cache simulator, but theirs is capable of analyzing the data cache also. To do so they maintain cache state vectors across blocks, and use those to resolve which possible virtual address ranges that will be in the cache. Stappert and Altenbernd [49] simulate the execution of basic blocks while maintaining two sets for each block; one for active cache blocks upon entry and another for the active blocks upon exit from the block. This works for both data and instructions.

Another feature that affects the overestimation of the results is branch prediction. Colin and Puaut [50] incorporate the effects of this feature by using a syntactic tree that allows them to place a bound on the number of erroneous branch predictions and thus bound the delay due to these. In an attempt to place more weight on measurement and less on modeling, Petters and Farber [51], [52] use path analysis to create measurement blocks. The object code is instrumented to force the execution of the path with the chosen block, and the measurements are taken from a real processor using a software monitor. Wolf and Ernst [53] also share the same idea, but they use a simulator of the processor rather than the actual device.

Finally, the feature that appropriately draws the largest amount of attention is processor pipelining. Pipelines can be modeled more easily than can caches thanks to the nature of instruction-level parallelism and its rather locally constrained effects. Lim *et al.* [41] use the same reservation tables as for the cache to capture conflicts in the use of the pipeline stages and data dependencies among instructions. All reservation tables are kept until timing information of preceding program constructs is known. Colin and Puaut [54] use a pipeline simulation function, PipeSim, to statically estimate the worst case time needed for a basic block. There are others who have geared their pipeline models to mimic those of available production processors: Stappert and Altenbernd [49] with the PowerPC; Wolf and Ernst [53] with the Strong-ARM; and the in-order superscalar SPARC I of Schneider and Ferdinand [55]. However, the issue with out-of-order superscalar machines is mainly one of overly complex behavior to be modeled statically. An example of this was shown by Lundqvist and Stenstrom [56], where they demonstrated that for dynamically scheduled processors, one cannot assume that the worst case instruction execution time necessarily corresponds to the worst case behavior. They propose code modifications to eliminate this discrepancy, but to do so they assume architectural support for explicit control of cache state. One way used to overcome this anomaly associated with these dynamic processors was to take measurements of basic block execution times using the actual processors [57], [58]. Alternatively, Burns and Edgar [24] propose the use of extreme value estimation to statistically model the variation induced by superscalar architectural features. Despite this, the best of these methods cannot estimate the effects of these features on each other, nor can they hope to feasibly capture all effects from local and global interactions.

At this point, having obtained a likely worst case path, and after having built a timing model for the instruction-level code, the step remaining is to compute the time needed to execute the program along this worst case path. Healy and Whalley's timing analyzer [59] uses control flow and constraint information to generate a set of paths where the range of iterations in one set do not overlap with other sets. This information is then combined with cache simulations and machine-dependent information to make timing predictions, choosing the longest time for the result. Stappert *et al.* [60] came up with a solution using timing graphs with nodes corresponding to basic blocks, annotated with results from pipeline/cache analysis stages. Lundqvist and Stentstrom also perform a path enumeration, but they operate at the instruction level [44]. A different methodology for WCET estimation is adopted by Colin and Puaut [61], who build a syntax tree representation of the program. Park and Shaw [62] employ an integrated approach that computes time estimates for atomic blocks. The timing technique then uses control loops with the time measurements and loop control, and a test loop that includes the program to be measured. The execution time is the difference between the measured times of the ontrol loop and the test loop. Bate *et al.* [63] used an approach that computes the worst case execution frequencies of Java virtual machine code for each program being analyzed, which is then combined with a step accounting for the target's dependent hardware information. Using this, however, they cannot account for instruction dependencies.

The use of implicit path enumeration techniques terminology can be initially credited to Li and Malik's method [64]. To build onto their use of linear constraints, they take the conjunction of the functionality and structural constraints to form a combined constraint set of linear equations that are solved by an integer linear program (ILP) solver. Puschner and Schedl [65] modified the ILP solver to accept their T-graph representation of programs, in which the nodes represent control flow, and the edges signify code segments weighted by execution times of these segments. Engblom and Ermedahl [66] extend this approach to accept context specifications using scopes and markers to obtain tighter bounds.

B. Performance Analysis by Design of Experiments

We propose the use of DOE [67] to build statistical models of software execution time. DOE provides the foundation for experimenters to systematically investigate hypotheses about systems and processes. By outlining a series of structured tests in which well-thought-out adjustments are made to the controllable inputs of a process or system, the effects of these changes on a predefined output or response variable can be observed and their significance evaluated.

There have been some attempts at approaching the problem from a different perspective. In [68], Edgar and Burns applied the Gumbel distribution to a random sample of observations drawn from their simulations. This serves as a limiting distribution for that independent sample. Hence the user can identify the confidence level they wish to achieve in their schedule and pick the value of the distribution which corresponds to that level. This will undoubtedly produce some loose bounds for the WCET. Bernat *et al.* [69] devised a scheme using execution profiles (EPs) to associate a piece of code with a representation of the relative frequencies with which particular events occur. Those events whose frequencies are kept in the EP are the different execution times that some code may require. Their work focused on how to use extreme value statistics to combine these piecewise EPs and arrive at an execution time biased towards the worst case and even pessimistic. They then further developed their work to create the tool support necessary to actually perform probabilistic worst case execution time analysis (the tool was named pWCET) [70]. A hybrid WCET analysis framework was suggested by Colin and Petters [71] and by Kirner *et al.* [72]. Such a solution will use the path analysis phase of static analysis to extract the path of longest execution, and then combine that with runtime measurements of the execution blocks rather than the static processor models used by the initial solutions. The advantages of doing so include the ability to capture the behavior of the processor and its various states of other components. These methods cannot deal with interactions across boundaries of different blocks and simply take execution time by constantly running the simulation over and over, and finally taking the maximum time observed.

Although DOE is widely used in both experimental science and manufacturing, it has seen little use in embedded software or computer architecture. The complex interactions between separate system-wide components demand a formal approach to gathering, analyzing, and identifying significant factors influencing a system's behavior, and the use of those factors in quantitatively characterizing that behavior. We believe that the application of statistical tools, like DOE, which use empirical evidence to reject hypotheses about the imputed efficacy of the characterization of systems, is a valuable system design tool. Statistical methods not only identify design problems but also help optimize the observed system using the remaining characterizations.

The following process contains a brief checklist of the crucial decisions that must be made at each phase of, and the flow through, the experiment. They are not independent, and amendments can be made iteratively [67].

- Identify the experiment goals.
- Name the expected causes of variation. This includes the treatment factors (i.e., the parameters to be changed) and their levels, the covariates that are observable but not controllable, and the blocking factors used to block the experiments into separate sets.

- Choose the method used to assign the experimental units to the levels of the treatment factors. The standard method is to do this randomly.
- Define the measurements to be made and the experimental procedure or setup.
- Run a trial experiment first. This is necessary to determine potential areas of difficulty.
- Designate the statistical model. This will represent the expected relationship between the response variable and the sources of variation from above. The techniques available for analysis later on will also depend on this.
- Outline the analysis. All hypothesis and tests to be made should be declared in order to ensure whether the data and model are sufficient.
- Calculate the number of observations needed. This is also to guarantee validity of tests and to minimize the required resources.

The analysis of variance (ANOVA) allows us to distinguish whether a given treatment (a set of design parameters, for example) has statistically significant effects.

Using the analysis performed by ANOVA, there are a host of other tests that the user can apply. These include asking for contrasts between specific levels of the treatment only, or requesting for contrasts comparing the average of a group of responses versus another group. Linear and quadratic trends in the treatment effects can also be estimated. For the case of multiple tests, one can perform all pairwise comparisons, or compare all treatment means with a control variable, or do the same with the best treatment.

We propose an iterative process encompassing the design and analysis principles established by DOE. This iterative procedure would work on producing fine-grain incremental refinements to the WCET estimate until achieving the target. The DOE utilization will prove its value when the decision needs to be made as to how to modify future iterations.

Fig. 6 shows our DOE-based software performance methodology. The initial step of generating random number values for the program inputs serves a twofold purpose. For this stage of the analysis, each input is considered as a treatment factor, and each randomly selected value of the input is considered as a level of that respective factor. This analysis allows us to model the variability of the effects of all the possible levels in the factor without needing to produce inputs for every possible value of the variable. The role of the user at this juncture is to analyze the results of the random effects to determine which inputs will be maintained during further analysis and which ones can be discarded. While that decision can obviously be made in software, the more critical role of the user remains to designate which type of analysis needs to be performed in the following stage. Since at this point the variance in output data can be verifiably attributed to the difference in input values, we can now proceed to establish the manner by which each input affects the response and what is the exact nature of its effect on the response. Depending on the process being analyzed and the type of treatments specified, the user will also determine the stopping criteria of the iterative stage later on. The next step chooses the treatment factors and their levels. These differ from the factors used in the initial random setup in that they do not symbolize a one-to-one correspondence with the task inputs; rather, they represent the types of treatments applied to the chosen inputs in an attempt to pinpoint the desired worst case settings. Their levels are the differing degrees to which the treatments can be used.

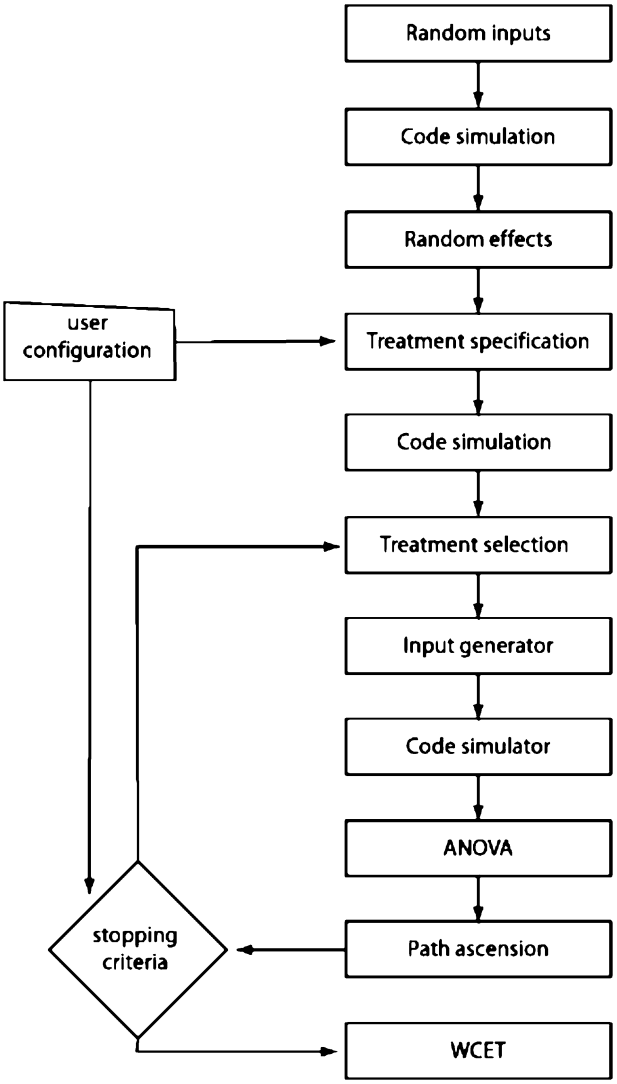


Fig. 6. ***DOE-based software performance estimation methodology.***

Once the treatment factors and their levels have been chosen, we can proceed with the main iterative process. The input generator depends on the treatment factors and their levels to create the proper stimulus for the experiments. Prior to entering this loop, we have already

determined which inputs significantly affect the response and the ways in which they do so. An additional confirmation can then be made by carrying out an analysis of variance to determine whether our assumptions of influential treatments were the correct ones to begin with. Now, having fixed the factors and their levels, we can systematically apply them to manipulate the inputs, generating test vectors for the process, which will lead us on the path of observing increasing execution time.

Each set of simultaneous inputs generated must be run through the simulator and subject to analysis. The ANOVA package determines which levels of the factors have the most impact on our response within the bounds of our current test. A lack of impact can be established for two different situations in our case here: if the factor exhibits no significant differences between any of its levels; and where some of the treatment levels effect on the response is significantly less than the effect of other levels. Hence, when choosing these factor levels wisely, we are able to modify the inputs in the direction of greatest ascent, and can continue to do so until we reach the limits of our coverage or until the response stops growing.

It is simple to determine when the expected response has ceased increasing; however, the bounds of the current test coverage area are resolved by examining the initial input value against the permissible factor levels. For example, if an input is a scalar integer and we determine the critical factor to be a scaling coefficient with fixed levels, then when we reach the maximum or minimum value to which the integer can be scaled, those levels those become the bounds of the coverage area. At this point, we might have either reached our maximum expected response value, or just the limits of our current experiment, and the stopping criteria as defined by the user will be the determining factor in this issue. Some examples of a stopping criterion can simply be a fixed number of iterations; or, for more accuracy, we can utilize the statistics again to test for when a desired number of consecutive iterations resulted in no significant increase in the expected response value. All that remains now is to either continue on with the loop passing on the state of the previous iteration to the treatment selection stage for further input generation or exiting with the result. The following sections describe the application of this method to some example code, and some of the results of the analysis and conclusions drawn from these experiments.

The framework needed to perform the experiments presented here was built in the Matlab environment. The simulator used in these experiments is the open source model SimpleScalar, simulating the ARM instruction set along with all of the possible architectural features found in modern processors, such as cache behavior, the pipeline, branch prediction, and even out-of-order issue.

We chose to test the proposed methodology on three tasks from the same general class of algorithms, namely, that of sorting. The three chosen were insertion sort, from the normal linear class of sorting algorithms; quick sort, from the divide and conquer algorithm category; and finally heap sort to represent the other class of sorting procedures. This class of routines was chosen for the depth of analytical work already available for them as a means to verify the experimentation procedure in use here, in other words, the worst case inputs have been identified and can be used for comparison purposes.

The first step is to feed the simulator the executables along with the random valued variables for input. In all three cases, the identified input variables were the input arrays to be sorted. In each experiment performed, the WCET estimate sought was for the execution of one of the programs with a fixed size array as input. The amount of input arrays produced by the random input generator were four times the chosen array size, where each element in the vector is chosen randomly, hence limiting the possibility of choosing two similar arrays. The other measure used to minimize the likelihood of that occurring was the large range allowed for each element. Each separate input was then designated as a level of the variable factor.

Table 1 displays the results for the random effects analysis performed for an array size of 64 elements. This test and all subsequent tests were performed with the significance level α set to 0.05; this parameter specifies what level of effect is necessary to be meaningful. The results for each different algorithm are referred to by its name under the source column. The other columns from left to right are sum of squares, degrees of freedom, and mean square, respectively. We want to determine whether the critical

Table 1 Initial Random Effects Analysis for Arrays of Size 64

Source	Sum square	Degrees of freedom	Mean square	F	Pr > F
Insertion	6987112.2	63	110906.5	48.26	0
Error	294179.3	128	2298.3		
Total	7281291.5	191			
Heap	1456966.6	63	23126.5	12.6	0
Error	234969.3	128	1835.7		
Total	1691935.9	191			
Quick	2074437.7	63	32927.6	17.42	0
Error	241889.3	128	1889.8		
Total	2316327	191			

Table 2 Two-Way Random-Effects Full Model Results

	Source	Sum square	Degrees of freedom	Mean square	F	Pr > F
Insertion	Order	161049.8	4	40262.4	19.06	0.0072
	Scale	1428.3	1	1428.3	0.68	0.4517
	Order*scale	35091.3	20	1754.6	1.2	0.3401
	Error	35091.3	20	1754.6		
	Total	206017	29			
heap	Order	8611.1	4	21652.8	11.43	0.0184
	Scale	1241.6	1	1241.6	0.66	0.4636
	Order*scale	7577.5	4	1894.4	0.54	0.7048
	Error	69544.7	20	3477.2		
	Total	164975	29			
quick	Order	33991.1	4	8497.78	19.51	0.0069
	Scale	43.2	1	43.2	0.1	0.7685
	Order*scale	1741.8	4	435.45	0.27	0.8913
	Error	31787.3	20	1589.37		
	Total	67563.5	29			

value of the F-test is greater than the ratios of the mean sums of squares.

Here in all three cases, results confirm the near impossibility of the event that a Type I error occurs, where we could commit a "false positive" or, in essence, choose to reject the null hypothesis that the different input arrays have no effect on the execution time when that is in fact true. Thus we can say with all confidence that the input array does have a significant effect on the measured response time.

Given that verification, we now need to provide the procedure with the treatment factors that can represent the sources of that input's effect on the response. Identifying the fact that a fixed size array's only source of change is either in the ordering of the elements with respect to one another or a scaling of the elements, we create two random effect factors—"order" and "scale"—to measure the significance of these types of changes on the response variable. The random values assigned to the factors are each considered one of the infinitely possible levels of that factor, and the designated analysis portion of this stage of the procedure becomes a two-way random-effects full model. The results corresponding to that analysis are listed in Table 2, with the first set of tests done for the insertion sort, the second for quick sort, and the third for the heap sort. Again, we can immediately examine the p-values for evidence of significant effects. Bearing in mind that we are striving for $\alpha = 0.05$ level significance, then any test reporting a p-value greater than that will not be considered significant enough to reject the null hypothesis. The third term here is the scale/order interaction term; it is a test of whether the effect of order/scale on the response variable changes with the changing values of the other one. If the interaction terms displayed significance, then we would not be able to adjust the input variables independently of one another. For all three cases, the only term displaying any significance is the "order" treatment factor.

At this point, we discard the other negligible terms and focus on a factor to treat the level of ordering in the array. Figs. 7–9 show the results of order factor tests for insertion sort, quicksort, and heapsort, respectively. In these figures, the *y*-axis labeled as the order factor levels represents the degree of order in the array. The factor levels signify decreasing order in the array as the factor level increases from one upwards. The lines on the plot imply significance for any two centered lines that have no overlap; then there is a significant difference at α-level of 0.05 between the expected execution time of those factor levels. Since we already know that this is the major source of variance in the response variable, we focus on it. When we allowed the procedure to run on the different algorithms for varying array sizes, for both insertion sort and quick sort, the worst case input was the reverse-sorted array, confirming what we know about the two algorithms.

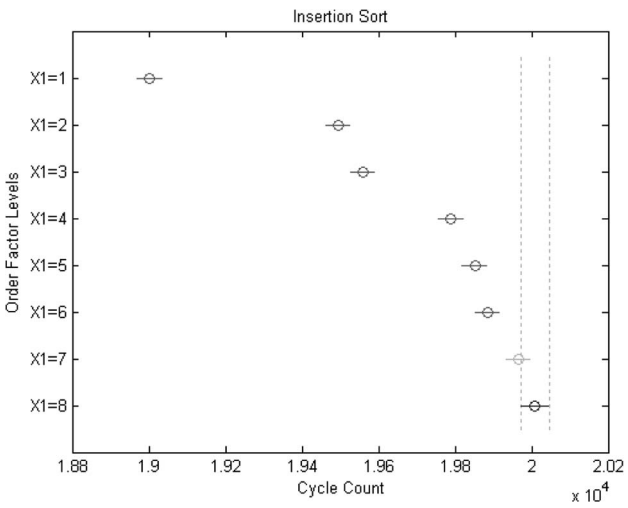


Fig. 7. ***Order factor test for insertion sort.***

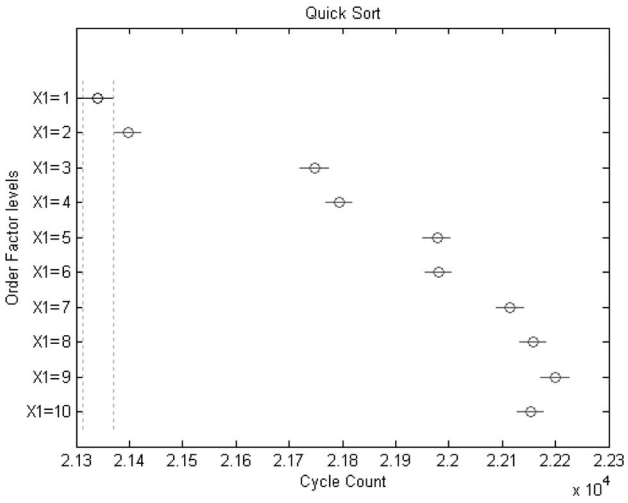


Fig. 8. *Order factor test for quicksort.*

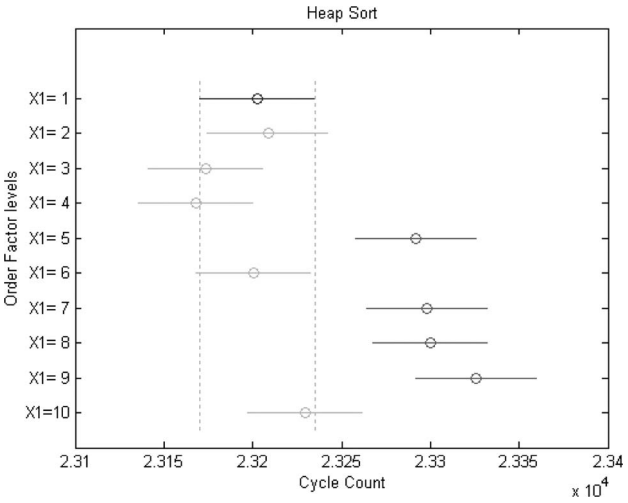


Fig. 9. *Order factor test for heapsort.*

IV. CONCLUSION

Modern hardware/software codesign methodologies must take into account a wide variety of both technical and nontechnical constraints. Technical constraints from control law considerations to Internet security. Nontechnical constraints on vehicular electronics include long product lifetimes and reliance on independent suppliers of components and subsystems. Traditional hardware/software codesign methodologies have been adapted to the requirements of vehicle design. Hardware architectures are built largely from predefined components and subsystems, which are then customized with software. Because most vehicle designs rely on software for most system functions, it is very important to determine the functional and performance-correctness of software modules. The growth of Internet-enabled vehicles puts new demands on system designers who want to capture the benefits of Internet connectivity (access to information, online diagnostics, etc.) while keeping the vehicle safe from online attacks. We expect that hardware/software codesign methodologies will continue to evolve as electronics components advance in complexity and the demands on vehicular systems continue to evolve. ■

REFERENCES

[1] W. H. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, pp. 967–989, Jul. 1994.

[2] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Des. Test Comput.*, vol. 10, pp. 29–41, Sep. 1993.

[3] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test Mag.*, vol. 10, pp. 64–75, Dec. 1993.

[4] C. M. Christensen, *The Innovators Dilemma*. Cambridge, MA: Harvard Business School Press, 1997.

[5] C. H. Fine, *Clock Speed: Winning Industry Control in the Age of Temporary Advantage*. New York: Basic Books, 1998.

[6] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[7] J. Backus, "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, Aug. 1978.

[8] G. R. Hellestrand, "MODAL: A system for digital hardware description and simulation," in *Proc. 4th Int. Conf. Hardware Description Lang.*, Palo Alto, CA, 1979, pp. 131–137.

[9] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[10] N. G. Leveson, *Safeware*. Reading, MA: Addison-Wesley, 1995.

[11] RTCA, Inc., "Software considerations in airborne systems and equipment certification," RTCA/DO-178B, Dec. 1992.

[12] National Research Council, *An Assessment of Space Shuttle Flight Software Development Processes*. Washington, DC: National Academies Press, 1993.

[13] D. A. Peled, *Software Reliability Methods*. New York: Springer, 2001.

[14] G. F. Franklin, J. D. Powell, and A. Emami-Naeni, *Feedback Control of Dynamic Systems*. Reading, MA: Addison-Wesley, 1986.

[15] *The Spin Model Checker—Primer and Reference Manual*, Addison-Wesley, Reading, MA, Sep. 2003.

[16] P. R. Gluck and G. J. Holzmann, "Using spin model checking for flight software verification," in *Proc. 2002 IEEE Aerosp. Syst. Conf.*, Big Sky, MT, Mar. 2002.

[17] S. Owre, N. Shankar, and J. Rushby, PVS: A Prototype Verification System From CADE 11, Saratoga Springs, NY, Jun. 1992.

[18] A. Galdino, C. Muñoz, and M. Ayala, "Formal verification of an optimal air traffic conflict resolution and recovery algorithm," in *Proc. 14th Workshop Logic, Lang., Inf. Comput.*, 2007.

[19] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math. Aspects Comput. Sci.*, vol. 19, J. T. Schwartz, Ed., Providence, RI, Dec. 1967, vol. 19, pp. 19–32.

[20] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.

[21] P. Cousot, R. Cousot, J. Feret, A. Miné, D. Monniaux, L. Mauborgne, and X. Rival, "The ASTRÉE analyzer," in *Proc. 14th Eur. Symp. Program. (ESOP 2005)*, vol. 3444, S. Sagiv, Ed., Edinburgh, Scotland, UK, Apr. 4–8, 2005, vol. 3444, pp. 21–30.

[22] E. Feron and F. Alegre, "Control software analysis, Part I and II," Tech. Rep. arXiv: 0809.4812, Oct. 2008.

[23] M. Rinard, "Credible compilation," Lab. for Computer Science, Massachusetts Inst. of Technology, Cambridge, MA, 1999.

[24] R. Kirner and P. Puschner, "Classification of WCET analysis techniques," in *Proc. IEEE Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, Seattle, WA, May 18–20, 2005, pp. 190–199.

[25] R. Chapman, A. Burns, and A. Wellings, "Integrated program proof and worstcase timing analysis of spark ada," in *Proc. ACM Workshop Lang., Compiler, Tool Support for Real-Time Syst. (LCRTS '94)*, Jun. 1994.

[26] P. Altenbernd, "On the false path problem in hard real-time programs," in *Proc. IEEE 8th Euromicro Workshop Real-Time Syst.*, L'Aquila, Italy, Jun. 12–14, 1996, pp. 102–107.

[27] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *Euromicro J. Syst. Architect.*, vol. 46, no. 4, pp. 339–355, Feb. 2000.

[28] T. Lundqvist and P. Stenstrom, "An integrated path and timing analysis method based on cycle-level symbolic execution," *Real-Time Syst.*, vol. 17, no. 2–3, pp. 183–207, Nov. 1999.

[29] A. Ermedahl and J. Gustafsson, "Deriving annotations for tight calculation of execution time," in *Proc. 3rd Int. Euro-Par Conf. Parallel Process.*, 1997, pp. 1298–1307.

[30] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise wcet determination for a real-life processor," in *Proc. First Int. Workshop Embed. Software (EMSOFT 2001)*, Jan. 1, 2001, pp. 469–485.

[31] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley, "Bounding loop iterations for timing analysis," in *Proc. IEEE Real-Time Technol. Applicat. Symp. (RTAS '98)*, Denver, CO, Jun. 3–5, 1998, pp. 12–21.

[32] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. V. Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Syst.*, vol. 18, no. 2–3, pp. 129–156, May 2000.

[33] C. Healy and D. Whalley, "Tighter timing predictions by automatic detection and exploitation of value-dependent constraints," in *Proc. IEEE Real-Time Technol. Applicat. Symp. (RTAS'99)*, Vancouver, BC, Jun. 2–4, 1999, pp. 79–88.

[34] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Syst.*, vol. 18, no. 2–3, pp. 249–274, May 2000.

[35] N. Holsti, T. Langbacka, and S. Saarinen, "Worst-case execution-time analysis for digital signal processors," in *Proc. EUSIPCO Conf. (X Eur. Signal Process.)*, Sep. 2000.

[36] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst, "Intervalbased analysis of software processes," in *Proc. ACM SIGPLAN Workshop Optim. Middleware Distrib. Syst.*, Snowbird, UT, 2001, pp. 94–101.

[37] F. Wolf and R. Ernst, "Execution cost interval refinement in static software analysis," *Euromicro J. Syst. Architect.*, vol. 47, no. 3–4, pp. 339–356, Apr. 2001.

[38] J. Engblom, P. Altenbernd, and A. Ermedahl, "Facilitating worst-case execution times analysis for optimized code," in *Proc. IEEE 10th Euromicro Workshop Real-Time Syst. (EWRTS '98)*, Berlin, Germany, Jun. 17–19, 1998, pp. 146–153.

[39] R. Kirner and P. Puschner, "Transformation of path information for wcet analysis during compilation," in *Proc. IEEE Euromicro Conf. Real-Time Syst. (ECRTS '01)*, Delft, The Netherlands, Jun. 13–15, 2001, pp. 29–36.

[40] C. Healy and D. Whalley, "Tighter timing predictions by automatic detection and exploitation of value-dependent constraints," in *Proc. IEEE Real-Time Technol. Applicat. Symp. (RTAS '99)*, Vancouver, BC, Canada, Jun. 2–4, 1999, pp. 79–88.

[41] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim, "An accurate worst case timing analysis for RISC processors," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 593–604, Jul. 1995.

[42] S.-K. Kim, S. L. Min, and R. Ha, "Efficient worst case timing analysis of data caching," in *Proc. IEEE Real-Time Technol. Applicat. Symp. (RTAS '96)*, Brookline, MA, Jun. 10–12, 1996, pp. 230–240.

[43] Y.-T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," in *Proc. IEEE Real-Time Syst. Symp. (RTSS '96)*, Los Alamitos, CA, Dec. 4–6, 1996, pp. 254–263.

[44] T. Lundqvist and P. Stenstrom, "An integrated path and timing analysis method based on cycle-level symbolic execution," *Real-Time Syst.*, vol. 17, no. 2–3, pp. 183–207, Nov. 1999.

[45] G. Ottosson and M. Sjodin, "Worst-case execution time analysis for modern hardware architectures," in *Proc. ACM SIGPLAN Workshop Lang., Compiler, Tool Support Real-Time Syst. (LCRTS '97)*, 1997.

[46] C. Ferdinand, F. Martin, and R. Wilhelm, "Applying compiler techniques to cache behavior prediction," in *Proc. ACM SIGPLAN Workshop Lang., Compiler, Tool Support Real-Time Syst. (LCRTS '97)*, Jun. 1997, pp. 37–46.

[47] F. Mueller, "Timing predictions for multi-level caches," in *Proc. ACM SIGPLAN Workshop Lang., Compiler, Tool Support Real-Time Syst. (LCRTS '97)*, 1997, pp. 29–36.

[48] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon, "Timing analysis for data caches and set-associative caches," in *Proc. IEEE Real-Time Technol. Applicat. Symp. (RTAS '97)*, Montreal, PQ, Canada, Jun. 9–11, 1997, pp. 192–202.

[49] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *Euromicro J. Syst. Architect.*, vol. 46, no. 4, pp. 339–355, Feb. 2000.

[50] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Syst.*, vol. 18, no. 2–3, pp. 249–274, May 2000.

[51] S. P. G. Farber, "Making worst-case execution time analysis for hard realtime tasks on state of the art processors feasible," in *Proc. IEEE Int. Conf. Real-Time Comput. Syst. Applicat. (RTCSA '99)*, Hong Kong SAR, China, Dec. 13–15, 1999, pp. 442–449.

[52] S. Petters, "Bounding the execution time of real-time tasks on modern processors," in *Proc. IEEE Int. Conf. Real-Time Comput. Syst. Applicat. (RTCSA '00)*, Cheju Island, Dec. 12–14, 2000, pp. 498–502.

[53] F. Wolf and R. Ernst, "Execution cost interval refinement in static software analysis," *Euromicro J. Syst. Architect.*, vol. 47, no. 3–4, pp. 339–356, Apr. 2001.

[54] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based wcet analysis," in *Proc. IEEE 13th Euromicro Conf. Real-Time Syst. (ECRTS '01)*, Delft, The Netherlands, Jun. 13–15, 2001, pp. 37–44.

[55] J. Schneider and C. Ferdinand, "Pipeline behavior prediction for superscalar processors by abstract interpretation," in *Proc. ACM Workshop Lang., Compilers, Tools Embed. Syst. (LCRTS '99)*, vol. 34, ser. *ACM SIGPLAN Notices*, May 1999, pp. 35–44.

[56] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. IEEE Real-Time Syst. Symp. (RTSS '99)*, Phoenix, AZ, Dec. 1–3, 1999, pp. 12–21.

[57] S. P. G. Farber, "Making worst-case execution time analysis for hard realtime tasks on state of the art processors feasible," in *Proc. IEEE Int. Conf. Real-Time Comput. Syst. Applicat. (RTCSA '99)*, Hong Kong SAR, China, Dec. 13–15, 1999, pp. 442–449.

[58] S. Petters, "Bounding the execution time of real-time tasks on modern processors," in *Proc. IEEE Int. Conf. Real-Time Comput. Syst. Applicat. (RTCSA '00)*, Cheju Island, Dec. 12–14, 2000, pp. 498–502.

[59] C. Healy and D. Whalley, "Tighter timing predictions by automatic detection and exploitation of value-dependent constraints," in *Proc. IEEE Real-Time Technol. Applicat. Symp. (RTAS '99)*, Vancouver, BC, Canada, Jun. 2–4, 1999, pp. 79–88.

[60] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *Euromicro J. Syst. Architect.*, vol. 46, no. 4, pp. 339–355, Feb. 2000.

[61] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based wcet analysis," in *Proc. IEEE 13th Euromicro Conf. Real-Time Syst. (ECRTS '01)*, Delft, The Netherlands, Jun. 13–15, 2001, pp. 37–44.

[62] C. Y. Park and A. Shaw, "Experiments with a program timing tool based on source-level timing schema," *IEEE Trans. Comput.*, vol. 24, pp. 48–57, May 1991.

[63] I. Bate, G. Bernat, G. Murphy, and P. Puschner, "Low-level analysis of a portable java byte code WCET analysis framework," in *Proc. IEEE 7th Int. Conf. Real-Time Comput. Syst. Applicat. (RTCSA '00)*, Cheju Island, Dec. 12–14, 2000, pp. 39–46.

[64] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1477–1487, Dec. 1997.

[65] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times—A graph based approach," *Real-Time Syst.*, vol. 13, no. 1, pp. 67–91, Jul. 1997.

[66] J. Engblom and A. Ermedahl, "Modeling complex flows for worst-case execution time analysis," in *Proc. IEEE Real-Time Syst. Symp. (RTSS '00)*, Orlando, FL, Nov. 27–30, 2000, pp. 163–174.

[67] A. Dean and A. Voss, *Design and Analysis of Experiments*. New York: Springer-Verlag, 1999.

[68] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling," in *Proc. IEEE Real-Time Syst. Symp. (RTSS '01)*, London, U.K., Dec. 3–6, 2001, pp. 215–224.

[69] G. Bernat, A. Collins, and S. Petters, "Wcet analysis of probabilistic hard real-time systems," in *Proc. IEEE Real-Time Syst. Symp. (RTSS '02)*, Austin, TX, Dec. 3–5, 2002, pp. 279–288.

[70] G. Bernat, A. Colin, and S. Petters, "Pwcet: A Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Syst.," University of York, U.K., Tech. Rep. YCS-2003-353, Jan. 2003.

[71] A. Colin and S. Petters, "Experimental evaluation of code properties for WCET analysis," in *Proc. IEEE Real-Time Syst. Symp. (RTSS '03)*, Cancun, Mexico, Dec. 3–5, 2003, pp. 190–199.

[72] R. Kirner, P. Puschner, and I. Wenzel, "Measurement based worst-case execution time analysis using automatic test-data

generation," in *Proc. Int. Workshop Worst-Case Execution Time Analysis (WCET '04)*, Catania, Italy, Jun. 29–Jul. 2, 2004, pp. 67–70.

[73] C. A. Wargo and C. Chas, "Security considerations for the e-enabled aircraft," in *Proc. IEEE Aerosp. Conf.*, Big Sky, MT, Mar. 2003, vol. 4, pp. 4-1533–4-1550.

[74] P. Polishuk, "Automotive industry in europe takes the lead in the introduction of optical data buses," *Wiring Harness News*, Nov. 2001.

[75] P. Koopman, J. Morris, and P. Narasimhan, "Challenges in deeply networked system survivability," in *NATO Adv. Res. Workshop Security Embed. Syst.*, Aug. 2005, pp. 57–64.

[76] J. Ray and P. Koopman, "Data management mechanisms for embedded system gateways," in *Proc. Depend. Syst. Netw. Conf. (DSN'09)*, Estoril, Portugal, Jun. 2009, pp. 175–184.

[77] C. Szilagyi and P. Koopman, "A flexible approach to embedded network authentication," in *Proc. Depend. Syst. Netw. Conf. (DSN'09)*, Estoril, Portugal, Jun. 2009, pp. 165–174.

[78] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem—Overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, Apr. 2008.

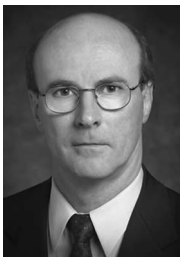
ABOUT THE AUTHORS

Ahmed Abdallah received the B.S. degree in computer engineering from the University of Maryland, College Park, in 2002 and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 2005 and 2008, respectively.

His research interests are in working on developing methodologies that employ empirically based approaches to create a structured process for designing and optimizing embedded system platforms. He is currently working at Embedded Systems Technology, San Carlos, CA, on the same subject.

Eric M. Feron received the M.S. degree in computer science from Ecole Polytechnique and Ecole Normale Supérieure, France, and the Ph.D. degree in aeronautics and astronautics from Stanford University, Stanford, CA, in 1990 and 1994, respectively.

Following an engineering appointment with the Ministry of Defense, France, he was on the faculty of the Massachusetts Institute of Technology in the Department of Aeronautics and Astronautics from 1993 to 2005. He is currently the Dutton-Ducoffe professor of aerospace software engineering at the Georgia Institute of Technology, Atlanta. He is a co-author of the monograph *Linear Matrix Inequalities in System and Control Theory* (SIAM, 1994), and the English translation of Bézout's famous *General Theory of Algebraic Equations* (Princeton, 2006). His research interests are the application of computer science, control and optimization theories to important aerospace problems, including flight control systems and air transportation.



Graham Hellestrand (Fellow, IEEE) received the B.Sc. (Hons.), Ph.D., and Exec. M.B.A. degrees from the University of New South Wales, Australia, and the M.B.A. degree from the University of Sydney, Australia.

He is Founder and CEO of Embedded Systems Technology, San Carlos, CA, his third startup. He is also Emeritus Professor of Computer Science and Engineering, University of New South Wales. He has published more than 100 papers in international conferences and journals, and is the principal author of two patents.

He is a Fellow of the Institution of Engineers, Australia. He is a member of the Australian Government's Information Technology Industry Innovation Council and a Director of National ICT Australia Ltd (NICTA). He has held a number of Board positions in IEEE CAS between 1994 and 2010. He has lived in Silicon Valley for the past 12 years.

Philip Koopman (Senior Member, IEEE) received the B.S. and M.Eng. degrees in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1982. After serving as a U.S. Navy submarine officer, he received the Ph.D. degree in computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1989.

During several years in industry, he was a CPU designer for Harris Semiconductor and an embedded systems researcher for United Technologies. He joined the Carnegie Mellon faculty in 1996. His research interests include dependability, safety-critical systems, distributed real-time embedded systems, secure embedded systems, and embedded systems education.

Prof. Koopman is a Senior Member of the ACM.

Marilyn Wolf (Fellow, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1980, 1981, and 1984, respectively.

She was with AT&T Bell Laboratories from 1984 to 1989. She was on the faculty of Princeton University from 1989 to 2007. She is currently Farmer Distinguished Chair and Georgia Research Alliance Eminent Scholar at the Georgia Institute of Technology, Atlanta. Her research interests include embedded computing, embedded video and computer vision, and VLSI systems.

Prof. Wolf is a Fellow of the ACM and an IEEE Computer Society Golden Core member. She has received the ASEE Terman Award and the IEEE Circuits and Systems Society Education Award.