INVITED PAPER

Platform-Centric Self-Awareness as a Key Enabler for Controlling Changes in CPS

This paper addresses the challenges in managing the continuous change and evolution of CPSs and their operation environment. It presents two frameworks, controlling concurrent change (CCC) and information processing factory (IPF), for building self-aware CPSs that have the capabilities of self-modeling, self-configuration, and monitoring.

By Mischa Möstl, Johannes Schlatow, Rolf Ernst, Nikil Dutt, Ahmed Nassar, Amir Rahmani, Fadi J. Kurdahi, Thomas Wild, Armin Sadighi, and Andreas Herkersdorf

ABSTRACT | Future cyber–physical systems will host a large number of coexisting distributed applications on hardware platforms with thousands to millions of networked components communicating over open networks. These applications and networks are subject to continuous change. The current separation of design process and operation in the field will be superseded by a life-long design process of adaptation, infield integration, and update. Continuous change and evolution, application interference, environment dynamics and uncertainty lead to complex effects which must be controlled to serve a growing set of platform and application needs. Self-adaptation based on self-awareness and self-configuration has been proposed as a basis for such a continuous in-field process. Research is needed to develop automated in-field design methods and tools with the required safety, availability, and security guarantees. The paper shows two complementary use cases of self-awareness in architectures, methods, and tools for cyber–physical systems. The first use case focuses on safety and availability guarantees in self-aware vehicle platforms. It combines contracting mechanisms, tool based self-analysis and self-configuration. A software architecture and a runtime environment executing these tools and mechanisms autonomously are presented including aspects of self-protection against failures and security threats. The second use case addresses variability and long term evolution in networked MPSoC integrating hardware and software mechanisms of surveillance, monitoring, and continuous adaptation. The approach resembles the logistics and operation principles of manufacturing plants which gave rise to the metaphoric term of an Information Processing Factory that relies on incremental changes and feedback control. Both use cases are investigated by larger research groups. Despite their different approaches, both use cases face similar design and design automation challenges which will be summarized in the end. We will argue that seemingly unrelated research challenges, such as in machine learning and security, could also profit from the methods and superior modeling capabilities of self-aware systems.

KEYWORDS | Cyber-physical systems; design automation; embedded systems; self-awareness

Manuscript received July 26, 2017; revised May 2, 2018; accepted May 25, 2018. Date of current version September 14, 2018. This work was supported by the Marie Curie Actions of the European Union's H2020 Programme, Deutsche Forschungsgemeinschaft (DFG) under Grants FOR1800, ER168/32-1, and HE4584/7-1; and by the U.S. National Science Foundation (NSF) under Grant CCF-1704859. *(Corresponding author: Mischa Möstl.)*

M. Möstl, **J. Schlatow**, and **R. Ernst** are with the Institute of Computer and Network Engineering, Technische Universität Braunschweig, Braunschweig 38106, Germany (e-mail: moestl@ida.ing.tu-bs.de; schlatow@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de).

N. Dutt, **A. Nassar**, and **F. J. Kurdahi** are with the Center for Embedded & Cyber–physical Systems, University of California at Irvine, Irvine, CA 92697-2620 USA (e-mail: dutt@uci.edu; anassar@uci.edu; kurdahi@uci.edu).

A. Rahmani is with the Center for Embedded & Cyber–physical Systems, University of California at Irvine, Irvine, CA 92697-2620 USA, and also with the Institute of Computer Technology, Technische Universität Wien, 1040 Vienna, Austria (e-mail: amirr1@uci.edu).

T. Wild, **A. Sadighi**, and **A. Herkersdorf** are with the Chair for Integrated Systems, Technische Universität München, Munich 80290, Germany (e-mail: thomas.wild@tum.de; armin.sadighi@tum.de; herkersdorf@tum.de).

Digital Object Identifier 10.1109/JPROC.2018.2858023

I. INTRODUCTION

A future cyber–physical system (CPS) will host a large number of coexisting distributed applications on hardware platforms with thousands to millions of networked components communicating over open networks. These distributed applications will include critical tasks, such as road-traffic control involving communicating autonomous cars and infrastructure, or smart energy controlling the energy grid down to the individual device. Often, distributed applications follow common design objectives, such as energy-efficiency, and guarantees for high availability, real-time or safety.

Such CPS reach far beyond classical embedded system design processes controlled by a single owner. They are subject to permanent change, environment dynamics and application interference. Applications using self-adaptation or machine learning dynamically change their properties and their resource requirements. Resulting short adaptation cycles of CPS applications would introduce system dynamics never experienced in the history of electronic design automation (EDA) before. Given the rapidly growing number of such CPSs and applications, there would not be enough engineering, service and maintenance personnel for user directed integration.

Communication has long adapted to this development by standardizing protocols and dynamically adapting the networks and resource assignment to changing user requests. From this perspective, the Internet-of-Things appears as a natural extension of the approach taken in communication. However, CPS design goes further, addressing complex side effects of such an approach on distributed applications. Even more so, the approach used in communication counters the established design processes for safety-critical and high-availability systems that assume static design processes organized in predefined, well-structured steps from concept all the way to in-field maintenance, and require highly predictable behavior as a basis of worst case guarantees. These requirements are formulated in safety standards, e.g., the generic standard IEC 61508 [1], the automotive standard ISO 26262 [2], or the avionics standard DO 178C [3].

To comply with these requirements, designers partition the system into noncritical and critical parts. The design idea behind this approach is to keep the critical parts static, which is currently a prevalent approach for mixed-criticality systems [4]. The approach resulting from the partitioning is to apply methods for static systems to the critical part of the system. Among these methods are techniques that promise strong isolation, such as static time driven scheduling [5, Ch. 10] or static system configurations as, e.g., in AUTOSAR. Static time slicing—a specific form of time-driven scheduling—is, for example, applied in the ARINC 653 standard for avionics equipment [6]. It dictates that, if a CPU shall be shared among software with heterogeneous certification requirements, the partitions for the software must be fully isolated through static time-slice scheduling. In the ideal case, this is possible, e.g., if software from different time partitions does not communicate. However, implementing ARINC 653 systems, e.g., on multi-core architectures where software communicates across cores, is already a complex issue. Challenging design issues arise if software with dynamic behavior, which according to the applied design paradigm must be assigned to the noncritical domain, must communicate with critical applications in the static part. While static budget assignments (e.g., time-slicing) might be feasible for systems deployed and maintained by an original equipment manufacturer (OEM) as the "single owner" (e.g., aircrafts) if all variables are known in sufficient detail, it is much more complicated in multi-owner systems. In any case, ARINC 653 and AUTOSAR both do not provide answers on how change—i.e., change of software, hardware, goals, paramaters, environment, etc.—can be managed under the constraints introduced by multi-ownership and dependencies such as communication between static and nonstatic system parts.

Hence, handling change is a first class problem in future CPS. The current separation of design process in the lab and operation in the field will be superseded by a lifelong design process of adaptation, in field integration, and update. Like most other system design processes, safety standards use the V-model to structure the system design process [2]. While the left branch of the V defines the design and implementation steps from requirements to implementation, the right branch defines the test and integration steps. The objective is to migrate part of the right branch to the field, such that in-field integration can incrementally accommodate changes. This does not necessarily mean that in-field integration becomes an online design process, but it requires right-branch design automation which can be applied in the field. Unfortunately, the right V branch is less automated than the design steps on the left. Important methods like the Failure Mode and Effects Analysis (FMEA) [7] are currently executed by the user with little formal and tool support, much like the integration verification that uses simulation and often has to resort to prototyping due to lack of complete and accurate models from the design steps. So, the EDA challenge of this approach is to provide a suitable model basis and appropriate algorithms and methods to support in-field integration.

For clarification in this paper, we distinguish several terms for methods executed in the field. At-runtime methods are executed while an application under change is active, i.e., concurrent to the application. They are also called online methods. At-downtime methods are executed while the application under change is not active. All these methods are subsumed under in-field methods. In-field methods have in common that they become part of the overall system function, in contrast to traditional lab-based methods following a separated design process. Here, we avoid the term "design-time" usually denoting all activities before deployment, because in this paper, design is extended beyond deployment.

A possible foundation for in-field EDA methods is the concept of self-awareness as will be explained in the next section. After the discussion of self-awareness in Section II, we will elaborate on two different use cases for the application of self-awareness in CPS design and maintenance. The first use case in Section III elaborates on the controlling concurrent change (CCC) approach, which aims to automate functional changes. This use case focuses on application change and integration. It relies on model-based contracting methods as well as automated analysis of platform interdependencies, and is exemplified based on an automotive systems example. The second use case is introduced in Section IV. It is a concept for efficient, autonomous, and adaptive multiprocessor system-on-chip (MPSoC) platform control under aging and changing operating conditions which shall enable long lifetime for mixed criticality applications. We present cyber–physical system-on-chip (CPSoC) as an exemplar self-adaptive MPSoC with a sensor-actuator-rich platform deploying a closed-loop paradigm emulating large-scale CPS. This presented example addresses the interplay of load profile, platform physics and model-based platform self-control in an information processing factory (IPF). In our conclusion in Section VI, we will utilize the use cases to summarize topics for EDA research into self-aware CPS.

II. SELF-AWARENESS

Self-awareness is well known in computing systems and includes a wide-range of capabilities, such as adaptivity, autonomy, self-modeling, etc. In particular, many research fields have been using self-awareness or related concepts in the past such that a variety of definitions can be found for this. This section therefore clarifies our generalized understanding of self-awareness and introduces it as a key concept for controlling changes in CPS.

A. Status

More than a decade ago, self-awareness was proposed for autonomic computing. In their seminal 2003 paper [8], Kephart and Chess from IBM envision "computing systems that can manage themselves given high-level objectives from administrators." Required capabilities already included self-configuration, self-optimization, self-healing, and self-protection against defined security attacks. Autonomic computing addresses large scale computer systems, as used in enterprise computing which are continuously controlled and maintained by humans, the administrators. Autonomic computing supports administration by automated diagnosis and offloads from detailed knowledge of functions and dependencies. Since then, many new contributions extended the role of maintaining these systems in an application-centric way. Applications interact with networked computing systems using control-theoretic mechanisms to optimize quality of service (QoS) for a variety of objectives. Self-awareness was used in the formation of virtual platforms supporting systems integration, increased system dynamics, and openness [9] preparing the basis for today's big data applications.

Meanwhile, this principle of self-awareness can be found in many research fields from control engineering [10] and autonomic computing [8] to traffic psychology [11]. Hence, many definitions of self-awareness can be found in literature, emphasizing different aspects of the targeted contexts [12]. We therefore resort to a very basic definition that generalizes the overall concept.

Definition 1: Self-awareness is the ability (of a computing system) to recognize its own state, possible actions and the result of these actions on itself, its operational goals, and its environment (including physics).

A major step in the development of self-aware solutions was the extension to networked embedded systems which are not supervised by humans. New paradigms, such as organic computing were proposed emphasizing self-control based on objectives derived from complex models of self-awareness or leading to emergent behavior [13], [14]. Physical system properties, such as media quality, automatic control quality, or energy consumption were added to account for embedded systems functions and resource constraints [15], [16]. Most recently, self-awareness in computing systems has been described as a shift from a reactive to a proactive operation that is achieved by a model-based loop of learning, reasoning and acting as an architectural concept [17]. In this context, group-awareness is defined as the ability of a subject to distinguish between itself, the environment and the peer group. The latter is treated differently by associating it with peer-group-specific expectations and goals [18].

Most of the previous work, however, proposes specific self-aware solutions for individual applications. In-field application integration and networked system control require platform-centric self-awareness that is flexible enough to handle very different types of applications. In the following we propose such an approach.

B. Self-Awareness as a Modeling Basis

Definition 1 assumes that system self-awareness requires an implicit or explicit self-model that not only describes its current state but also the possible next states. In a manual design and maintenance process, this knowledge is encoded in design documents and in the competence of the integrator. Migrating parts of the integration process to the field requires formalization of this knowledge to derive appropriate, formally applicable methods. In practice, documentation is not complete, might be partly imprecise, and uncertain. Due to dynamics, the data change over time.

With self-awareness, self-modeling becomes an independent EDA research goal. This includes methods for setting up, maintaining and verifying a self-model under the concerns of uncertainties and inaccuracies.

Definition 2: Model uncertainty is a set of structural properties and parameters of a model that are undecidable or ambiguous.

As an example, consider thread mapping on a multicore processor. If the scheduler decides thread assignment at runtime, the model cannot ultimately specify on which core of the processor the thread will execute. Furthermore, the speed of a core may vary, if power-saving techniques such as frequency scaling are applied—hence the worst case execution time (WCET) of a task can vary depending on the power-state of the processor.

In self-modeling, models of different design aspects can be combined. The fusion of such partial models can reduce uncertainties or help to detect inconsistencies. This can include physical sensory data as in the IPF example in Section IV. Such "uncertainty management" can help to improve design efficiency. Model semantics can be extended to handle different kinds of uncertainties from sensor data, possible errors in sensor interpretations, or data with limited trustworthiness.

While uncertainty is part of the model semantics, inaccuracy describes the deviation of the model from physical reality.

Definition 3: Model inaccuracy is the result of insufficient approximation of a model to the realities,

for instance, the inappropriate application of a linear model to nonlinear behavior. Such inaccuracies can arise from unobservable state changes, modeling errors or deviations, or undetected software errors. Besides these safety relevant aspects, inaccuracies can also originate from intentionally incorrect models, which may become a security issue. There are many approaches to detect and often quantify inaccuracies: experimental plant characterization known as system identification in control theory [19], execution time monitoring [20] or profiling, to name a few which are applied in the use cases as follows. Handling model inaccuracies is a necessity: while possible inaccuracies could generally be captured as model uncertainties, the resulting large uncertainty bounds will make design with such general models unfeasible.

Fig. 1 shows how we structure self-modeling for CPS w.r.t. three different aspects. The function self-representation models the coexisting and interacting applications and their logical architecture. The platform self-representation models the platform components and their interaction while the physical representation models the physical environment and the physical properties of the CPS. This structure was chosen to reflect the world of existing EDA models and shall open the door for applying EDA methods in the field. However, in current design practice, these models are used in separate verification steps, usually based on simulation and/or prototyping. By applying these models in completely separate phases, it is only required that the models represent the same static design. Yet, using unrelated models is not sufficient for self-awareness, as changes in one model must be reflected in the other models to know the consequences on the system state, actions and environment (cp. Definition 1). Although there exist techniques such as sandboxing [21] which constrain these consequences, they can only cover part of the possible effects—they are certainly helpful but not sufficient. Instead, we require knowledge about relations between objects of different models in order to establish a coherent and self-aware view on a CPS. Note that this can be achieved by applying a holistic modeling methodology where the models follow a clear abstraction hierarchy, which is sufficient but not necessary for model coherence. More precisely, model coherence does not require a strict specialization/abstraction relation as found in holistic meta-models; the necessary and sufficient property is that related models are free of conflicting information, i.e., are coherent.

Hence, automated in-field integration requires more formal methods and dependencies between models become crucial. As an example, an FMEA for distributed CPSs (see Section III) can reveal dependencies between functions extracted from a combination of function and platform models. Such a dependency can arise from timing interference on a processor core or from a clock frequency that is influenced by the ambient temperature. Therefore, new in-field methods and respective EDA tools require model coherence. The two use cases of this paper exploit coherence between two model types each, as shown in Fig. 1.

The goal of self-awareness in the use cases is to ensure self-modeling and monitoring to maintain an accurate image of the system state. Through these self-awareness capabilities automatic design steps are possible, that would traditionally be lab-based and carried out by engineers. Our goal in both use cases is to move automated steps into the field. The in-field EDA tasks are then subject to the same verification and validation as the traditional lab design, with the only difference that it is performed automatically. In the following two use cases, we focus on different challenges when applying (platform-centric) self-awareness to enable in-field EDA for CPSs: The CCC use case follows an analytical approach with contracting, dependency analysis, formal safety analysis, and monitoring for handling functional changes. It primarily focuses on dealing with model uncertainty in both (functional and platform) worlds. The challenge here is to automatically maintain coherent models that capture and monitor safety and other nonfunctional properties of the entire

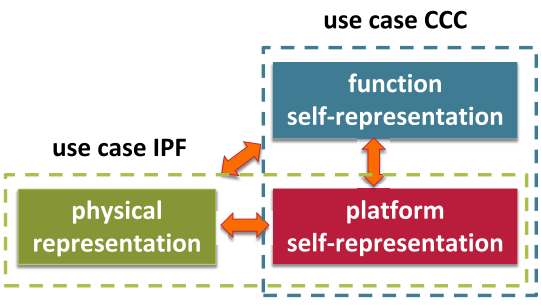


Fig. 1. *Self-representation in CPS.*

system that result from platform sharing. The IPF use case relies on incremental changes and feedback control of (physical) effects of the platform, i.e., system identification and model coherence from platform models upward is the primary goal. Dealing with the complexity of modeling physical effects and the involved uncertainties and inaccuracies becomes a major challenge in this use case. With respect to self-awareness, both approaches are challenged by the platform complexity of CPSs (e.g., heterogenity, architectural complexity) as well as the ubiquitous nonfunctional requirements in this domain. The latter must not be neglected but rather dealt with at design time and by self-awareness in combination, i.e., at design time, the room and boundaries (design corridor) should be specified and leave room for self-awareness to cope with uncertainties and inaccuracies at runtime.

III. SELF-AWARE ARCHITECTURE FOR CONTINUOUS CHANGE AND EVOLUTION

Updates for commodity embedded systems, such as smartphones, have nowadays become an established technique. They either bring new applications to the device or change existing applications or functionalities. Updating has become an integral part of the life cycle and is done while devices are in the field and in the possession of consumers. The enabler for this is the shift towards deploying the devices as an open platform rather than a closed product such that third-party updates do not need to be channeled through the device's OEM. On the contrary, designing and deploying updates and new applications for other embedded systems and especially CPSs remains a difficult and challenging task, which is addressed by the German DFG Research Unit CCC (eight faculties, six years funding) at TU Braunschweig.

When trying to answer the question, why continuous updates and lifetime (software) maintenance are rarely done for embedded systems that carry out safety-critical functions, one has to take a look at the design process of these systems and their applications. The classical V-model as depicted in Fig. 2 is the prevalent method to structure the design process of these systems. In the V-model, every system—which is a set of functionalities, carried out by a defined set of applications—is designed by starting with a requirements specification at the left top of the V. During traversal of the V, these are refined on the descending branch and culminate in the hardware and software implementation at the bottom of the V. On its ascending branch, integration of individual functions and subsystems takes place such that extensive testing or validation can be applied to verify whether the initial requirements are met.

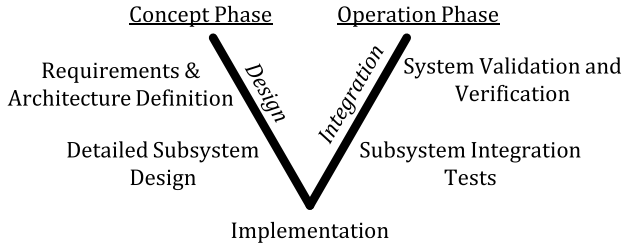


Fig. 2. *V-Model for system-level development.*

The V-model design object can range from a single application to a complete vehicle integration with many interdependent applications. In the context of safety-critical application domains, the understanding of system is rather holistic, taking the platform with all its functionalities into account. In other domains, it is also common to understand individual functions as the system under development, as for instance done with smartphone apps. In the latter case, as the individual functions are developed in isolation, such a process would require total isolation of functionalities on the platform, which is not achievable for CPS. This can be easily understood with the following example: Consider a vehicle as an example for a CPS. The functions of braking and accelerating for longitudinal guidance are two integral functions of any vehicle. Both require the wheels of the vehicle to carry out their task, i.e., they share elements of the CPS. Consequently, there must be a mutual exclusion or a strict hierarchy between the two functions to operate correctly, which is impossible to realize if the functionalities are to be totally isolated. As in the given example, many functionalities of CPSs can be safety critical, i.e., can cause harm to humans or their environment. Therefore, common safety standards, such as ISO 26262 in the automotive domain or IEC 61508 for general electric/electronics (E/E) or programmable-electronic (PE) systems, require the V-model as a system-level process. Individual function designs can also follow the V-process, but the overarching and holistic V-process for the system is essential, as—besides careful implementation—safety guarantees are a direct result of this process guiding the system's integration. In this process, methods like FMEA [7] or fault tree analysis (FTA) [22] are applied to reveal dependencies introduced through the implementation, as these can cause (harmful) interference between functions. Because of such side effects, the results of these methods are only valid for exactly one configuration/parametrization and allow no variances within the functions and their implementation. Furthermore, as a system is a static set of functionalities and their implementation, any update, upgrade or other change forms a new system. As a consequence, this new system has to undergo the complete process again if safety must be ensured for any element in the systems. Enforcement of properties that technically enable functional safety in the original system can certainly provide partial isolation, however, are not necessarily sufficient as it is not guaranteed that other introduced dependencies can undermine the isolation.

With respect to allowing in-field changes to safety-critical systems—which is one of the main goals of CCC—we therefore need to automate the essential steps of this design process. This particularly includes the steps

that enable, e.g., safety assurances as a property of the process. Hence, in-field changes, such as software updates, become an EDA problem as these changes must undergo the system-level design process to be integrated into the system. In the scope of CCC, we develop such an automated integration process that not only approaches safety requirements but also security, availability and real-time aspects in applications for the automotive domain and avionics. In this context, we further use the term *change* to consolidate all aspects of intended alterations of a system's platform or functional representation. Such changes can, for instance, not only originate from user input but also from a new operation environment that triggers a functional adaptation, or from a degrading platform.

In the remainder of this section, we elaborate on how this EDA problem is approached in the scope of CCC, and also provide insights into the architectural approach that enables the in-field application of such an EDA. In particular, we describe a modified design process that is tailored for a continuous life-cycle management of CPSs. We also elaborate on what particular methods are applied in order to automate design decisions and still provide strong guarantees on nonfunctional requirements such as safety and security. With the architectural approach in CCC, we further emphasize the role of self-awareness as a key concept to close the gap between model-based EDA methods and the actual CPS.

A. Cross-Layer Modeling and Dependency Analysis

As any EDA method relies on suitable modeling, we now summarize the important aspects of modeling for automated integration in the design process. We extend the idea of self-representation of systems introduced in Fig. 1 by a more detailed cross-layer model that picks up on modeling function as well as its platform. An example for a cross-layer model can be found in Fig. 3, which is formally represented as a graph. At the top layer of the functional model (light blue box), we find a logical architecture of the system, describing the high-level logical elements of a system (dark blue blocks). These logical elements map to individual functions (red circles) which are necessary to carry out the intended actions described by a logical element. This mapping is indicated in Fig. 3 by the blue dashed arrows between logical elements and functions. Interactions and dependencies between particular elements are expressed by solid black arrows. Note that one functionality can have mapping ties with more than one logical element and can thus be shared within the system, e.g., f_4 in Fig. 3 which is a shared function.

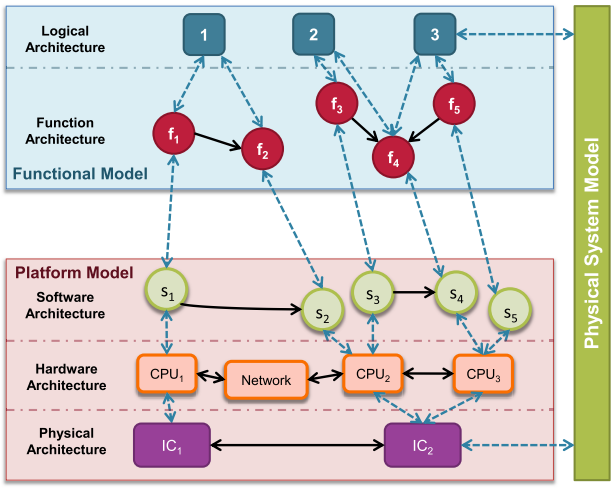


Fig. 3. ***Cross-Layer modeling.***

Although logical elements appear as independent elements in the logical architecture, coupling of these can already exist on the function layer by sharing a common function. In general purpose systems, changing and modifying logical elements of a system is comparatively easy, as all functional dependencies are known and can be tested in advance. This test can, e.g., follow the V-model process. However, in CPS design one faces a number of additional nonfunctional requirements such as functional safety, isolation for better security, real-time, reliability, availability, and many more. As these requirements are tied to platform- and implementation-specific knowledge, the architecture layers in the platform model from Fig. 3 provide a detailed view on these aspects. The software architecture describes the implementation of functions in software, as well as the interaction within the software system. The elements of the software architecture itself are deployed on several (logical) hardware elements (processors, memories, peripherals, etc.) as described by the mapping relations between software and hardware architecture. Again, the hardware architecture captures hardware dependencies. The mappings from hardware to physical architecture describe on which particular chip hardware elements are integrated.

We can see that each architecture layer within the functional model and the platform model has distinct properties that it describes and which are typically abstracted in hierarchical models. Nevertheless, the dependencies that are explicit in lower layers affect elements in higher layers, as dependencies are only *hidden* by abstraction but not eliminated, for instance, by sharing a common hardware element such as a processor (in the hardware architecture) or an operating system service/function (in the function architecture).

By introducing mapping relations between the architecture layers, a cross-layer model of the system is derived. This model enables tracking dependencies resulting from a multitude of effects on the different modeling layers. One can identify cross-layer dependencies that are not present in the architecture of the affected elements, i.e., the elements' architecture model suggests that both elements are independent. In order to do this, the Functional Model alone does not suffice, and possible dependencies within the implementation, hence the platform model,

must be considered as well. Consider the system sketched in Fig. 3. Dependencies either appear directly in an architecture model between two elements, or via mapping to an element in another layer. Consequently, also paths that traverse multiple layers can construct a dependency path for elements that have no direct dependency in their own architecture.

As a first example we illustrate that already the function architecture can procure dependencies that are abstracted on the logical architecture. On the Logical Architecture we see three independent elements in Fig. 3. Due to the mapping relations, we can, however, track that the logical element 2 is implemented by function f_3, which resorts to the shared function f_4. However, f_4 is also used by f_5, which creates a dependency between the two functions and thus also between the logical element 2 and 3. Further, and as a second example highlighting the importance of the Platform Model, we have to consider that the implementation of f_3 is realized in software component s_3. We can track a path in the graph given by Fig. 3 from s_3 via CPU_2, s_2, to f_2, which maps to logical element 1. In consequence, also elements 1 and 2 must be assumed dependent on each other, as long as no reasoning whether all dependencies are relevant has been performed. Note that, in order to assume two elements as independent under all relevant dependencies, no path of relevant dependencies must exist between them in the cross-layer model. This implies that, in the case of f_3, f_4, and f_5, from the first example we must also show that there is no relevant dependency between f_3 and f_5 on other architecture layers.

These examples highlight the necessity of cross-layer modeling as a prerequisite for exposing dependencies, that otherwise would be hidden by abstraction and possibly neglected. However, it also carves out that, besides exposing the dependencies, an evaluation of them is necessary due to their sheer number. Evaluation thus implies that dependencies must be quantified to be assessed as either relevant or negligible for a certain functional or nonfunctional property in the context of the system.

We refer to the discovery, quantification, and assessment of dependencies in the cross-layer model as dependency analysis. This requires for cross-layer modeling that coherency of the models must be ensured at any time, i.e., if change happens, the implications must be reflected in all respective models, and dependency analysis must assess the resulting cross-layer model.

While seemingly solving the dependency analysis problem, application of such a cross-layer model faces two main challenges. While individual layer models are usually available, at least in critical designs, the mapping relations between layers are often only known late in the design and are likely to be updated in case of changes on any of the levels. That is the rationale behind the holistic approach of fault analysis. In case of dynamic mapping, such as in case of task migration or dynamic scheduling in the runtime system, dependencies can even change at runtime. So, mapping and dependency analysis must be able to handle uncertainties and underspecified systems. This is addressed by defining sets of possible mappings and, hence, sets of possible dependencies [23]. The second challenge is relevance. When following all possible dependency paths in Fig. 3, it becomes obvious that all elements depend on each other. This is an overly pessimistic result which is useless for safety critical systems design. Therefore, the dependencies must be classified by relevance for the effects under analysis. Usually, such classification requires dependency quantification. Well known examples include error probabilities or timing interference. Today, relevance is determined by a human expert in a time-consuming and error-prone process. The current lack of appropriate models and automated methods is a main obstacle for a useful cross-layer design analysis and, consequently, for automated change management for safety critical systems.

For a design process, that particularly aims at incrementally changing a system, a suitable formulation of *change* is necessary. In the CCC approach, change requests are formalized that specify a modification based on models, i.e., every change request includes the "new" executable software along with a suitable model, describing it. An important prerequisite for software changes is a certain modularity and interchangeability. In software engineering, this is well-known as *separation of concerns*. More specifically, *component-based software engineering* is one approach that emphasizes this principle by composing a software system from components that solely interact over well-defined interfaces. The interfaces generalize from the actual implementation and therefore keep the components interchangeable.

In CCC, we aim for component-based models of CPS—including software as well as hardware components—as they reduce dependencies in the architecture to the explicitly modeled interfaces and thereby keep the dependency analysis tractable. The components are generic building blocks of the system that is composed from these components such that they implement the desired functionality and fit to the particular target platform. Each change to the system must be coherently representable in a system-wide model for analyzing any potential cross-layer dependencies, as well as for other analyses to ensure freedom from interference for the individual functions that a set of (software) components create.

B. Architectural Approach

In the scope of CCC, an architecture is developed that implements the aforementioned life cycle of CPSs, particularly with applications in automotive vehicles and space robots. This architecture is composed of two segregated domains: the model domain and the execution domain.

Fig. 4 depicts the resulting system architecture. The execution domain is based upon existing operating systems and runtime environments that execute on multiple (networked) platform components and host several application components. It is further augmented by application and platform shaping/monitoring capabilities

that allow parameter enforcement and the observation of the application and platform components at runtime. All changes to the execution domain are managed by the model domain, which is implemented by the multichange controller (MCC). The MCC therefore performs the integration of changes and deploys the corresponding configurations to the execution domain. The interface between execution domain and model domain is built on a contracting formalism. Contracts can be seen as sets of assumptions and resulting guarantees if the specified assumptions are fulfilled. In the CCC setup, contracts are negotiated between the model and the execution domain, i.e., if the execution domain's behavior complies with the model domain's assumptions, the system provides the negotiated guarantees.

In the ideal case, contracts are executed properly, i.e., the model assumptions are faithful and the execution domain behaves accordingly. However, this is an idealized view on systems, assuming complete and correct knowledge. In real systems, inaccuracies in the form of a deviation of model and corresponding observation are the norm. To cope with these inaccuracies, the interface of model and execution domain in Fig. 4 provides the model domain with metrics observed in the execution domain. This interface allows the model domain to interpret the metrics and detect deviations of actual behavior from model knowledge. Knowledge about deviations is the basis for adjusting models to make them coherent with observed behavior. This self-modeling ability is a prerequisite for self-awareness in the CCC system. To achieve self-awareness, the model domain must further be able to negotiate new contracts on the refined self-model of the system. Note that the execution domain's operation is independent from this refinement and negotiation whereas its configuration may only be changed at (function) downtime.

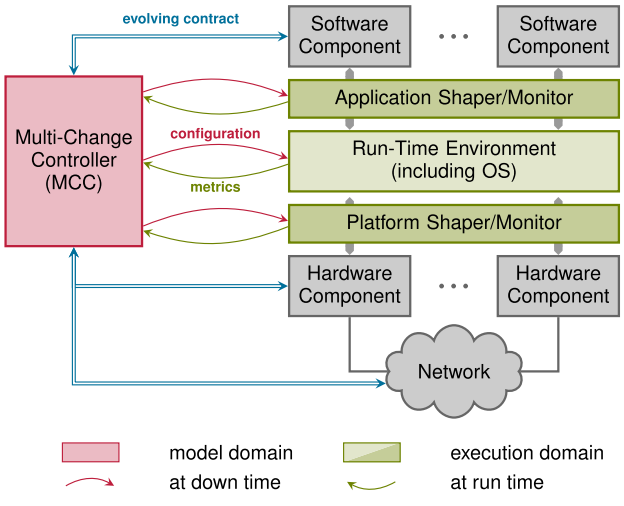


Fig. 4. *CCC architecture comprising a model domain (red), an execution domain (green) as well as changing software/hardware components (gray).*

1) Model Domain: The CCC model domain is implemented by the MCC (cf., Fig. 4), which takes full control over the configuration of the execution domain. Such a configuration specifies all relevant parameters that are necessary to set up the execution domain. In order to guarantee an uninterrupted operation, we only permit configuration changes at downtime of the execution domain, i.e., in a safe state. Nevertheless, the MCC may operate as a (budgeted) background process at runtime of the execution domain, as long as normal operation of the system is not impaired.

One objective of the MCC is to find a suitable configuration for a specified set of *integration contracts*. These contracts are a formalization of parameters and requirements of, e.g., a software or hardware component. More specifically, the MCC guarantees that all the contracted requirements (e.g., response time) are satisfied by the resulting configuration under the assumption that the components adhere to the specified parameters (e.g., execution time). In realistic systems, the contracted requirements will depend on parameters from other integration contracts. For instance, a response time constraint can only be guaranteed if the given execution times of all interferers are accurate and conservative. Cross-layer dependency analysis allows us to discover all relevant parameters that act as assumptions for a certain requirement. We formalize this relation as tuples of assumptions and guarantees and thereby create *derived contracts* which we use to establish the contracting interface between the model domain and execution domain. This contracting interface effectively closes the gap between models and implementation as it enables the MCC to react to changes in the assumptions. This becomes particularly relevant in the presence of model uncertainties and inaccuracies, for which we must question the quality of the provided guarantees.

In general, any change will trigger the MCC to perform an automated design process in reaction to the changed integration contract or derived contract. For this purpose, the MCC must keep a coherent cross-layer model of the supervised system covering software and hardware components, operating system (OS) and runtime environment (RTE), as well as shaping and monitoring mechanisms. As previously stated, we separate functional and platform aspects in the model (cf. Fig. 3) due to the fact that the functional model depends on the application domain and the platform model on the target platform.

For the platform, we apply a component-based and service-oriented model in which the platform hosts several software/hardware components that communicate via service-oriented interfaces. Given a functional model of the system, the MCC must decide which components need to be part of the system in order to integrate the modeled functionality and how these components must be connected. For this, we differentiate between *application components* and *repository components*: An application component implements (a part) of a particular function and can therefore be directly related to the functional

model. Repository components, on the other hand, perform more generic tasks such as relaying access to hardware devices or networks. Both types of components may require or provide certain services. The MCC must therefore solve pending service requirements—for instance by inserting repository components—in order to generate a consistent network of communicating components [24]. Note that nonfunctional requirements, such as memory and response-time requirements, will additionally constrain the solution space. The parameters of this solution space are not only the selection and interconnection of components but also the mapping of components to processors and scheduling parameters. The MCC must therefore make automated design decisions such that all requirements are satisfied and appropriate contracting interfaces can be established. For this purpose, we resort to an automated model transformation and refinement from implementation- and platform-independent models to implementation- and platform-specific solutions [25]. This process is assisted by *analysis engines* that provide expert knowledge about particular design decisions and check for constraint violations. As constraint violations might only be detectable on lower model layers, this process may require iterations by backtracking to the conflicting design decision. By applying a backtracking algorithm—which guarantees termination—for this design-space exploration, a solution is found in limited time if it exists. Although it may take a very long time to search the entire design space, we do not have any requirements for this as we focus on feasibility of the design automation rather than efficiency or even optimality. More precisely, as the MCC only performs EDA, we do not require the MCC to find a solution within a particular time limit or to find a solution at all. The analysis engines allow reusing established analyses and evaluating specific aspects in isolation without the need of a holistic view. This reflects the fact that diverse requirements arise in different application domains. For instance, such analyses may focus on security threats in vehicles [26], byzantine agreement in cooperating vehicles [27], optimized resource assignment for FPGAs [28], hard response-time constraints [29], [30], or weakly hard timing constraints [31]. Note that the design-space exploration is centrally managed to ensure consistency of design decisions from multiple analysis engines.

Furthermore, dependency analysis [23], [32] is employed as an overarching method, which serves two purposes. First, it separates relevant from nonrelevant dependencies and, secondly, it determines neuralgic points where relevant dependencies need to be contained. These neuralgic points either result from inappropriate design decisions or from model uncertainties and inaccuracies. If necessary for establishing a sound contracting interface, shaping mechanisms can be applied at those points in order to supervise and enforce particular contract parameters. As mentioned earlier, detected parameter changes are also fed back to the MCC in order to renegotiate the contracting interfaces. As the dependency analysis provides the information which parameters are influenced by a changed contract parameter, it enables an incremental consideration of these changes such that the MCC can react with a reconfiguration of the execution domain. In consequence, this supervision and feedback mechanism equips the system with self-awareness.

2) Execution Domain: As indicated above, the execution domain must build a solid foundation for dealing with in-field changes in terms of a continuous life-cycle management for CPSs. In this section, we give a more detailed account of the particular mechanisms and solutions to which we resort in regard to this.

The main purpose of the execution domain consists of providing the RTE (and OS) for the applications. This RTE must permit changes (*configurability*) and likewise allow the MCC to keep control over any changes. Ideally, such an RTE will already provide strong isolation between software components and enable fine-grained access control in order to enforce the modeled behavior.

With respect to configurability, we can differentiate three paradigms: static, dynamic and reconfigurable. With *static*, we denote systems whose configuration is set at compile time (e.g., typical AUTOSAR/OSEK systems). *Dynamic* systems, on the other hand, can be configured at runtime. Typical examples are general-purpose OSs (e.g., Linux) that allow the user to start arbitrary processes, which often requires rather lax access restriction. The third paradigm, *reconfigurable*, stands for systems whose configuration may only change at boot time or downtime. By design, this paradigm thus enables the best control over changes, which is an essential prerequisite for the CCC approach. Moreover, in order to apply the model-based methods, we must also have fine-grained control over access permissions such that dependencies can be restricted to a minimum. We therefore pursue the principle of least privilege, i.e., a white-list approach of access control that only grants the minimum set of required permissions to a software component. Note that, in general-purpose systems, this approach is usually hard to employ as it quickly impedes the usability. On the other hand, in conjunction with an automated integration/configuration process, it becomes manageable and a key concept for guaranteeing an explicit structure of runtime dependencies.

In order to control changes and access permissions by the MCC, CCC resorts to—but is not limited to—microkernel-based systems which allow both by design, fine-grained access control as well as reconfigurability. In these systems, software components are operating in distinct address spaces in order to establish a spatial isolation. Communication between these components is therefore made explicit for which the microkernels typically provide interprocess communication (IPC) and signaling mechanisms. This effectively mitigates side-effects between components as memory accesses and interactions are restricted

to the necessary extent (principle of least privilege). With this concept, microkernels intend to minimize the trusted computing base (TCB), i.e., the amount of code on which an application must rely. As the TCB not only comprises the kernel and the application itself but may also include other applications, minimizing the application-specific TCB becomes an important element of security-focused architectures [33]. A major achievement of these minimization efforts is that they enable the formal verification of microkernel implementations as in the case of seL4 [34] or MUEN [35]. Apart from spatial isolation, temporal isolation is another important aspect in CPS in order to provide freedom from interference. Common concepts are time partitioning as in ARINC 653 [36], [37] or budget-based scheduling [38], which have also been recently adapted by microkernel implementations [39], [40].

The main concept of microkernels is the separation of policy and mechanism, i.e., the kernel should be kept clean from any particular policy but only provide the pure mechanisms. In consequence, applying and enforcing policies becomes an issue of the RTE. In the scope of CCC, we particularly use the *Genode OS Framework* [41], which provides a strictly component-based RTE with service-oriented interfaces, and aims at minimizing the application-specific TCB. Component-based (operating) systems are a straightforward continuation of the microkernel approach [42] and particularly suitable for formal verification [43]. Note that a clean and homogeneous application/component model simplifies the EDA.

Detailed knowledge about the RTE becomes an important aspect when establishing the corresponding models. In particular, it must be known to which extent the defined policies are enforced by the RTE. Where necessary, the RTE is augmented with shaping mechanisms as a contract enforcement to strengthen the assumptions that can be made by the model domain. It also equips the execution domain with self-protection capabilities against potentially harmful behavior. For instance, shaping of interrupt frequencies or execution times may be required to safely bound the interference on critical tasks. As a contract-enforcement mechanism, shaping can also guarantee integrity of network communication [44], [45], enforce assumptions in the timing model [20], provide fault detection and recovery [46], [47], or even avoid unaccounted interference in wireless communication [48]. In consequence, shaping is the main technique to address model uncertainties.

Nevertheless, contract enforcement alone does not equip a CPS with self-awareness. It only becomes self-aware if we add a feedback loop to the model domain that allows self-assessment of the system's behavior. By this, the system can learn from observations, reason about necessary changes, and act/adapt upon these. Monitoring acts as supervision and observation of a managed system and is a well-known concept in self-aware architectures [49], [50] or self-adaptive systems [51], [52]. This monitoring is especially relevant for model properties that cannot be accurately modeled, such as the presence of security leaks for which anomaly detection has been shown to be a feasible countermeasure [53]. In consequence, runtime monitoring also serves as a technique to addresses model inaccuracies.

As the name suggests, runtime monitoring and shaping always come with a runtime overhead. However, event-tracing frameworks have already become a common infrastructure in microkernel RTEs [41], [54] and act as an enabler for the efficient instrumentation of component interactions at runtime.

C. Example

Having discussed the cross-layer modeling and architectural approach of the CCC project, this section illustrates these methods by means of an example. For this purpose, we have a look at the integration of an automotive subsystem that performs lateral and longitudinal guidance of a road vehicle. Lateral guidance primarily uses the yaw rate provided by an inertial navigation system (INS) in order to control and correct the steering and ensure stability of the vehicle. Optionally, it also uses a reference trajectory calculated by a camera-based lane detection as a secondary input. Longitudinal guidance controls the vehicle speed based on the measured acceleration which is provided by the INS based on accelerometers and wheel-speed sensors. As an optional secondary input, it uses radar-based distance detection that implements an adaptive cruise control (ACC).

Due to the similarity in their sensor usage, both functions are typically implemented in the same subsystem. From the CCC perspective, we suppose that both functions have been developed independently, e.g., by using software-engineering techniques such as interface- or service-based design. However, as lateral guidance can be considered more critical than longitudinal guidance [55], we must show independence of both functions in the scope of an automated model-based integration process. Furthermore, we must also show their independence from other functions that share or connect to the vehicle platform, e.g., infotainment, navigation, etc. In the remainder of this section, we give a more detailed account of how this can be achieved by cross-layer dependency analysis. We specifically focus on timing dependencies for brevity, however, handling other dependencies, especially physical ones, follows the same principle.

1) Subsystem Setup and Implementation: The conceptual software and hardware architecture for the subsystem is illustrated in Fig. 5. It shows the distributed implementation on two electronic control units (ECUs), a switched Ethernet network, and dedicated camera and radar modules; other components (software as well as hardware) of the vehicle platform are omitted for simplicity. The INS is implemented by three sensor components (red)—i.e., *Gyro.*, *Accel.*, and *WheelSpeed* —and the *INS* component. The *Gyro.* component performs sensor fusion of

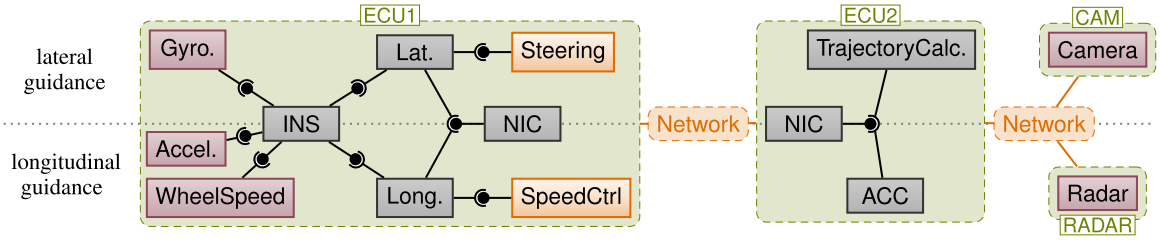


Fig. 5. ***Software/hardware architecture of the INS example with lateral and longitudinal guidance consisting of several interacting software components that execute on networked hardware components (ECU1, ECU2, CAM, RADAR).***

three gyroscopes for yaw, pitch and roll, while the *Accel.* component fusions the readings for three accelerometers in x, y, and y orientation. Further, the *WheelSpeed* component processes readings from the vehicle's wheel-speed sensors into current speed and acceleration. Resulting values from the three sensor components are reported to the *INS* component and aggregated. It subsequently provides the current angular velocity and linear acceleration as distinct services. The subscribed clients (*Lat.* and *Long.*) are notified whenever new values are available. Note that the gyroscopes are sampled with a much higher frequency than the accelerometers and wheel-speed sensors as the latter two are more precise and less affected by drift. A camera sends raw image data to ECU2 over the network. Similarly, the radar performs object recognition and also transmits the results to ECU2. On ECU2, trajectory calculation and ACC are performed based on the data received over the network, which is interfaced by the *NIC* (network interface controller) component. The results are periodically sent to ECU1.

On ECU1, the reference trajectory and target velocity is received and placed into read-only shared memory. Lateral guidance is then implemented by the *Lat.* component, which reads the *INS* and trajectory data from shared memory and calls the *Steering* component that provides an interface to the vehicle steering actuators. Similarly, longitudinal guidance (*Long.*) reads the *INS* and cruise control data from shared memory and calls the *SpeedCtrl* component to control the speed with the corresponding vehicle actuators (motor and brake).

As mentioned above, the lateral guidance is a critical function as it ensures stability of the vehicle and has respective safety goals associated with it [55]. The safety goals, e.g., are expressed in tolerable control overshoot, which is dominated by maximal tolerable dead time experienced by the control algorithm component. This can, for instance, be derived through functional analysis of the control algorithm. As dead time corresponds to reaction times of component paths, they can be specified together with the safety requirements in the integration contract describing the change and as a mapping in the supplied models.

2) Dependency Analysis: Considering the function in the context of a cross-layer model of the vehicle, it is obvious that numerous cross-layer dependencies exist. In the excerpt of the overall system in Fig. 5 alone, we can easily see that a number of software components map to the same ECU, and that the processing chains starting at the camera and radar sensors utilize the vehicle's network. The network is assumed to be shared with functions such as navigation and passenger entertainment. The latter are typical QM functions (not safety relevant), which must not interfere with functions that have an Automotive Safety Integrity Level (ASIL) requirement [2]. From the structure alone, all functions must be assumed transitively dependent, hence defeating the safety concept. As laid out above, the relevant dependencies must be distinguished from the nonrelevant ones.

Since the lateral guidance operates with a rather high frequency compared to other dynamic vehicle functions, it is sensitive to timing interference, i.e., the dependencies that possibly generate interference on the timing of the component paths must be evaluated for relevance. In order to detect and quantify any timing dependencies, the timing model (task graph) for ECU1 as depicted in Fig. 6 must be considered, since all interference paths with other functions traverse ECU1, as a common point of interference. The task graph models the ECU's workload by tasks

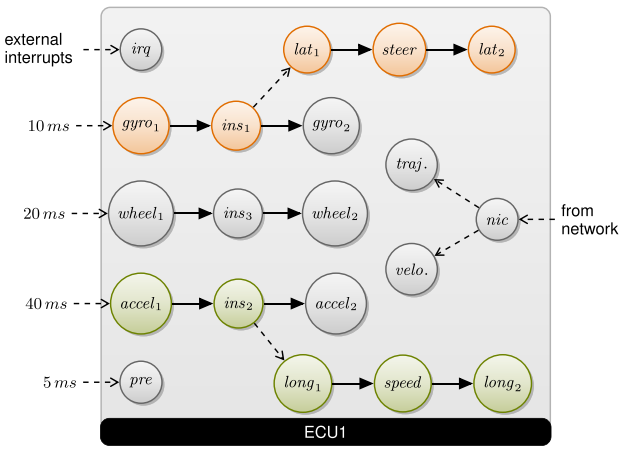


Fig. 6. ***Timing model for ECU1 in Fig. 5 illustrating the task chains for lateral guidance (orange) and longitudinal guidance (green).***

(circles) and their precedence constraints (solid/dashed arrows) and can be extracted from the software architecture given appropriate knowledge about the service interfaces [30]. Orange and green tasks in the figure represent the timing-critical part of the lateral and longitudinal guidance as two separate task chains. Note that the longitudinal and lateral guidance tasks are triggered from ins_1 and ins_2 respectively in order to process the updated values. Furthermore, the task graph contains a task chain indicated in gray in which the wheel-speed sensor processing and reporting to the *INS* is performed by the task ins_3. Note that each ins_x task represents a different path in the *INS* component, i.e., depending on which interface of the *INS* is used, a different functionality is provided. The remaining tasks can be considered as interference in this example as they result from sensor preprocessing (*pre*), network stack (*nic*), and other higher priority load (*irq*). To resolve the dependencies on the ECU we must investigate its scheduling.

As we use static-priority preemptive scheduling on this ECU, we must determine a priority assignment that satisfies both schedulability and freedom from (unbounded) interference. The latter gains relevance if we include model uncertainties such as the quality of WCET parameters. For critical functions (here: lateral guidance), we must ensure that every (timing) dependency can be safely bounded. This includes the task chain itself—as it contains service dependencies—as well as higher priority interference and lower priority blocking.

With respect to schedulability, a rate-monotonic priority-assignment (higher rate = higher priority) is optimal but potentially adds timing dependencies. In this example, we therefore employ a "criticality as priority" scheme that reduces timing dependencies by design but requires schedulability analysis.

As lateral guidance is the most critical function in this example, we assign its task chain the highest priorities and perform a response-time analysis in order to calculate an upper bound on the chain's latency [30]. However, the acquired bound is only safe if all parameters on which it is based are safe. In this example, we particularly notice that the tasks ins_2 and ins_3 appear as lower priority blockers to the lateral-guidance chain (orange) because they belong to the same software component and thus cannot be preempted by ins_1 (cf. [30]). Note that these tasks represent another path in the INS component and are not necessarily verified or carefully tested as is required for the criticality of the lateral-guidance function.

In consequence, three conditions must be ensured to mark timing dependencies on ECU1 as irrelevant: First, the required response times can be met with the assumed/specified execution rates and execution times so that timing guarantees can be given. The relevance threshold here is whether the response-time requirement can be met. Second, all parameters utilized for this conclusion are known with the same level of assurance, e.g., the parameters for ins_2 and ins_3 are known with the same level of confidence as those for the tasks in the orange chain itself. The threshold for this dependency is the assurance of the parameter, which typically depends on the method how the parameter is obtained. Third, other cross-layer paths these parameters depend on can be considered irrelevant as well.

Table 1 Contract Negotiation Results

contract	dependencies	iteration #0	#1	#2	#3
Response Time *lat*	**Worst-Case Execution Times (WCETs):** $gyro_1$, ins_1, $gyro_2$, lat_1, $steer$, lat_2, pre, IRQ, nic, $traj$, ins_2, ins_3 **Activation Patterns (ACTs):** IRQ, nic	○[1]	○[1]	○[1]	✓
WCET ins_2, ins_3	ECU1, WCET determination	—	✗[2]	✓[3]	
remaining WCETs	ECU1, WCET determination	—	○[4]	○[4]	✓
ACT IRQ	environment	—	✗[5]	✓[6]	
ACT nic	network	—	✗[5]	○[7]	✓
ECU1		—	—	✓[8]	
network				✓[9]	

[1] acceptable if contracts for WCETs and ACTs can be negotiated
[2] WCET determination process does not deliver the required assurance for the critical chain
[3] acceptable after applying shaping to guarantee WCET with the same assurance as for the critical *lat* chain
[4] acceptable if contract for ECU1 can be negotiated
[5] ACT controlled by external source
[6] acceptable by applying shaping to guarantee ACT
[7] acceptable if contract for network can be negotiated
[8] acceptable because hardware fault probability of ECU1 is below the required ASIL threshold
[9] acceptable because the corresponding network stream is controlled at the source and interfering streams are shaped in the network

Table 1 shows the results of four iteration steps (0–3) based on dependency analysis for ECU1. It indicates in which step a dependency can be accepted (✓), i.e., can be considered irrelevant. Furthermore, it also exemplifies that, to determine relevance thresholds, further contracts must be negotiated in order to accept a dependency (○). For instance in iteration 0, in order for the response time *lat* to be accepted, we need to negotiate the dependent contracts for WCETs and ACTs first. Moreover, if a contract cannot be accepted (✗), countermeasures must be applied. For instance the dependence on the Worst Case Execution Time analysis parameters for ins_2 and ins_3 cannot be considered irrelevant in iteration 1 as the process for determining the Worst Case Execution Time is not as stringent as for ins_1. It only becomes irrelevant in iteration 2 after shaping enforces an upper bound on the worst case execution time parameters.

Timing dependencies on the network and other hardware elements require similar steps: For the camera and radar data that share the network with other traffic, analysis also suggests that shaping mechanisms in the execution domain can be used to guarantee QoS even in the presence of unpredictable traffic, e.g., from brought-in devices that are granted access to the network. However, note that this only addresses the timing dependency, other cross-layer effects must still be assessed for relevance. For

instance, both guidance functions and the entertainment would depend on the same Ethernet network, i.e., the network and its configuration still pose as a single-point-of-failure. The relevance of this depends on the effects that are expected for the system and the failure rates that can be expected under the assumed operational conditions. For brevity, we only sketch the idea behind the further analysis here. If, e.g., transient hardware faults are to be expected, suitable fault-tolerance protocols can be applied and subsequently their timing implications included in the timing analysis [56]. On the other hand, for permanent hardware faults redundancy measures are necessary, i.e., the dependency is not acceptable as the rate for permanent faults is above the accepted threshold.

3) Contracting Aspects: Any configuration of the execution domain must satisfy the initially specified contract requirements of all functions in order to be valid. For this, it is required that all dependencies are classified in relevance and that no unacceptable relevant dependency remains. To achieve this, revealed dependencies are iteratively broken down into derived contracts such that their acceptance can be determined and negotiated.

In this example, we focused on the initially specified latency bound of the longitudinal and lateral guidance, which are supposed to be integrated as depicted in Fig. 5. The contracts that are necessary due to the dependency analysis' steps, can be derived from parameters that specify the integration, e.g., on which ECU components are implemented and how the priorities are assigned on ECU1 to perform a response-time analysis. Despite focusing on timing contracts in this example, other types of contracts need to be formulated in order to cover other types of requirements. Note that how to efficiently formulate and specify such contracts and system properties is an orthogonal problem and not in the scope of this paper.

Besides validating the configuration, the derived contracts in Table 1 implicitly select the metrics that can be evaluated by the model domain, and which shall be observed by the execution domain. The example highlighted that the model-based guarantees depend on the execution time of ins_2 and ins_3. Consequently, observing whether these model properties actually hold at runtime also checks the validity of the contracts between execution and model domain, and whether renegotiation is necessary to reduce shaping interventions.

This approach could even be extended such that certain integration parameters which become necessary during dependency analysis are automatically synthesized by the MCC. For instance the MCC could compute possible component deployments based on the given hardware platform and subsequently assigning priorities based on further dependency analysis steps.

D. Conclusion

CCC automates the integration process for CPS by copying many aspects of the existing engineering process, i.e., how and when design decisions are made, reasoning about requirements, etc. In CCC, the MCC implements this EDA methodology that replaces the lab-based integration and testing while the execution domain is kept self-contained (operational without the MCC) thereby equipping the CPS with the ability to apply design changes in place of the OEM. In particular, the CCC methodology exploits strong guarantees (contracts) that can be formulated for lab-tested software components and the microkernel architecture. For instance, our example illustrated how the integration of components for a safety-critical function is performed on these contracts and how requirements and dependencies are reasoned about. However, weaker guarantees (model uncertainty) can also be dealt with by using dependency analysis to expose neuralgic points and synthesize possible countermeasures. When it comes to model inaccuracy, runtime monitoring (and its feedback to the MCC) is essential.

IV. IPF

With decreasing feature size, semiconductor technologies expose higher parameter variation. In several national research priority programs, such as the NSF Variability Expeditions [57] or the German SPP Dependable Computing [58], new solutions have been developed to address these challenges, ranging from the circuit level all the way up to the software and applications layers. These solutions are applicable at design time, at downtime, at runtime or a combination thereof. On-chip sensors are used to identify the hardware health status including automatic control of current temperature or to mitigate aging effects. Redundancy in many forms has been proposed to guarantee continued performance, real-time constraints and integrity for safety critical and high availability embedded and cyber–physical applications. Sensor networks have profited from dynamic energy control which will support the pervasive use of very-large-scale integration (VLSI) in the up-coming Internet of Things (IoT). More and more of these on-chip control loops and higher level platform scheduling and management strategies have been employed to operate complex applications executing efficiently on an integrated circuit. Yet, these mechanisms (that span multiple levels of abstraction, and which address different goals) are largely uncorrelated and targeted to control individual effects. While acceptable for today's circuits and applications, the number and impact of effects to be controlled increases and requirements to dependability, resilience and longevity grow. Therefore, a holistic approach which covers VLSI circuit operation as well as runtime OS and application software is needed.

In this section, we introduce a concept that addresses these challenges and highlight how EDA can profit from control oriented self-aware platform management; we illustrated this with aspects from four examples of preliminary work and motivate the overall concept with a highly topical use case.

A. IPF Concept

Future microelectronic systems can be compared to factories. To keep factory production running effectively and efficiently, the production itself, the logistics of supply (material, energy, water, waste), the machinery and transport, the facility control including heating or illumination are all adapted to the current workload, while at the same time considering maintenance and continuous operation. Future microelectronic systems face comparable requirements. Much like in a factory, there will be a platform operation layer that controls the performance and health status of a microelectronic system based on an on-chip sensor network, considering the many different targets of temperature, energy, aging, reliability, security and long term systems evolution when it schedules the application functions, the memories, I/Os, and possibly micromechanics (where applicable). That platform operation layer will not only focus on the current status, but also predict the future state of a microelectronic system including the expected development of the platform. Where possible, it will communicate with other platforms to identify the development of the processing load and act accordingly. The platform layers will use their own IT infrastructure and reach a level of autonomy that is far beyond what is possible today.

Such holistically controlled autonomous microelectronic systems can be considered information processing factories (IP factories). The transformative vision is that these "IP factories" could become the dominant microelectronic platform beyond the current age of many core processors because of their flexibility, controllability, longevity, and evolution potential. They could adapt to modes of extremely low energy consumption and highest performance, as well as changes in other operational characteristics.

The IP factory is not only suitable to meet the challenges of future microelectronic systems, it is also a perfect platform for future autonomous systems. All major trends—networked embedded systems, CPS, systems-of-systems, or the IoT—assume higher subsystem autonomy to reach the required scalability. Due to its sensory capabilities, its flexibility and adaptability as well as the built-in autonomous control, an IP factory provides the conceptual basis for system level autonomy. Its capabilities can be utilized to enable self-awareness and context-awareness of the subsystem and to support system level collaboration for group awareness as well as to enable and control emergent system behavior. Together, component and system-level autonomy will form a hierarchy which separates concerns and appears to be eminently suitable for the design of large scalable systems.

Even though the IPF concept addresses all levels of CPS, the main focus is the "factory," i.e., the platform that processes information including its hardware and operation. The IPF concept is currently elaborated and investigated by an international research group of UC Irvine, TU Munich, and TU Braunschweig. The joint project is funded by the DFG and the NSF.

B. Objectives of Self-Aware Information Processing Factory

The central objective of a self-aware information processing factory in this specific context is the effective exploitation of self-awareness and self-organization in order to provision complex, MPSoC-based hardware-software CPS with a holistic information processing run-time control infrastructure for optimizing performance, power dissipation and system resilience.

The motivational metaphor of the information processing factory clearly indicates that bundles of component-specific, uncorrelated control instances are inadequate to orchestrate multicriterial objective functions of complex systems. Equally true, a strictly centralized controller for such systems has to fail due to lack of scalability. Therefore, a hybrid—as much modular and distributed as possible, as much centralized as necessary—and hierarchical approach must be developed that is viable, cost-/overhead-efficient and scalable.

In order to achieve the stated objective, the following conceptual design principles and mechanisms (see also Fig. 7) can be deployed. Sensor information will be tapped, fused and merged into SO/SA control processing entities at various abstraction levels of the hardware/software architecture of an MPSoC-based CPS system. The control

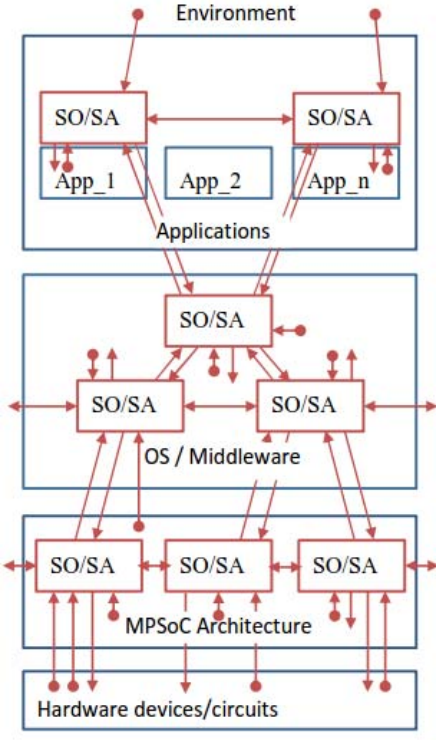


Fig. 7. *Self-organization/self-awareness (SO/SA) MPSoC stack.*

processing entities generate actuation directives to affect the MPSoC system constituents at the same or lower levels of abstraction. Individual control processing entities shall be delegated well-defined degrees of autonomy in operation [self-organization, self-awareness (SO/SA)] to locally optimize the entity-specific performance, power consumption and reliability. Correlation among multiple control processing entities, either at the same or among different abstraction levels, shall be established through the awareness for the "actuation-to-implication causalities" caused by an applied action (e.g., how an increase in frequency affects entity performance and power dissipation, or how a task scheduling policy affects energy consumption and application performance).

Self-organized, autonomous control, especially when applied by multiple hierarchical entities at different abstraction levels and with different control cycle periods, may lead to divergent or even contradictory control directives. Such forms of destructive emergence must be avoided. They can be detected by observing system interaction and controlled by monitoring and jointly enforcing global constraints and objectives. Self- and group-awareness provide a suitable modeling basis for the resulting globalized interaction and control. This is especially important in critical systems as will be discussed in the following.

The viability of such an approach can be assessed in terms of optimality of the achieved system performance, power efficiency and resilience relative to a state-of-the-art reference approach. Cost can be quantified in terms of fractional overhead of the SO/SA sense-control-actuate entities in comparison to the pure functional CPS hardware/software components. Scalability can be assessed in terms of numbers of architectural entities up to how many, as well as with how few, the approach is plausible.

C. EDA Perspective of IPF

Just as in the case of CCC, in-field methods for IPF extend current design steps to the field. But, the design goals change and so do the EDA tool challenges. The IC is no more optimized for the best operating point but it is optimized to efficiently adjust the operating point to changing operating conditions. This goal change affects verification and (online) test. It might be useful to extend testing by elements of system identification (see above) observing the influence of parameter change on circuit behavior under parameter change. EDA should support synthesis of control stacks for robust learning (cp. Fig. 7) using suitable architectures.

Furthermore, research on the self-aware IPF paradigm must address a whole host of fundamental problems that pose new challenges to EDA. How to ensure robust, predictable CPS system performance, power consumption and resilience through a combination of self-aware layers of autonomous, heterogeneous hardware/software (self-organization based) control on the one hand side and hierarchical top-down control resembling an IPF? How can self-awareness of individual system constituents or subsystems be transferred and combined into higher level group-awareness? What cost/overhead for a holistic SO/SA control infrastructure can realistically be achieved? Is it possible to bound this cost/overhead of the equivalent functional MPSoC layer complexity?

All these are open problems for architecture and related EDA research.

D. Preliminary Work on IPF

Although many related aspects have been explored earlier, they lack awareness and hierarchical, cross-layer autonomous management in MPSoCs through a relevant and comprehensive conceptual framework such as the proposed IPF. The following subsections provide an overview on preliminary works to IPF where self-awareness and self-adaptivity have been exploited at individual but different abstraction levels of MPSoC systems: autonomous system-on-chip (ASoC) uses hardware-based machine learning to control operation parameters of processor cores at the hardware architecture level, CPSoC introduces an adaptive and reflective middleware software stack for self-aware computing and communication control based on multi-layer sensing, and non-uniform verification architecture (NUVA) provisions a distributed runtime verification infrastructure utilizing self-replicating, low overhead runtime verification (RV) monitors and checkers.

1) ASoC Platform With Machine Learning-Based Control: ASoC platform [59], [60] deploys hardware and software reinforcement machine learning techniques (learning classifier systems) [61] on homogeneous multicore processors for optimizing workload balancing, power consumption and resilience against intermittent core failures in a self-aware/self-organizational manner. The general idea is that parts of today's and future abundant chip capacity, in form of MOSFET transistors, shall be dedicated for generic self-awareness/self-organization purposes in order to flexibly react to changing system and environmental operating conditions at runtime. The ASoC platform, as shown in Fig. 8, can be considered as a forerunner for the envisaged layered IPF MPSoC Architecture depicted in Fig. 7. The ASoC architecture augments the conventional functional or plant layer of a SoC by a so-called autonomic layer, consisting of interconnected autonomic elements (AEs). This separation of the ASoC architecture in two layers is only a conceptual view; physically, functional elements (FEs) and AEs will be realized intertwined on the same die. Preferably, FEs are existing IP library blocks such as CPU cores, NoC building blocks, on-chip memories, dedicated accelerators and I/O blocks. The idea is to use them untouched or "as is" in order to maximize reuse and preserve earlier investments in IP library development. Self-aware control and autonomous at-runtime optimization of key MPSoC operation parameters (e.g., CPU core supply voltage and frequency, task mapping to individual cores)

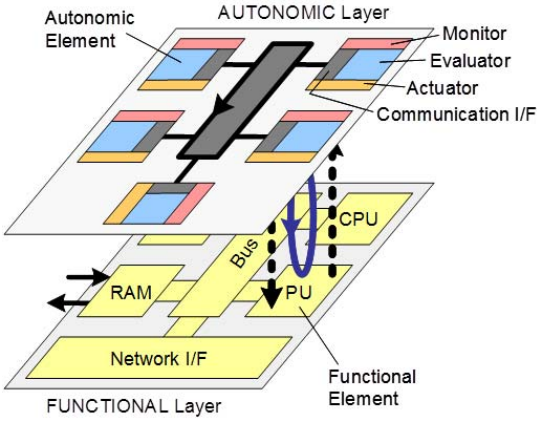


Fig. 8. *ASoC architecture layering.*

is achieved through closed "monitor-evaluate-act" control loops between individual FE-AE pairs.

In order to grant individual AEs awareness of the operation states in other AEs (and their FEs), AEs are interconnected at the autonomic layer through a dedicated ring network.

An AE consists of four building blocks: 1) the monitor, for collecting status information from the supervised FE; 2) the actuator, to issue FE operation parameter changes; 3) the communicator, to interface to the mentioned ring exchange network; and 4) the evaluator, to analyze the monitored status information and to propose potential actions. The evaluator obtains *self-awareness* of what is going on in its local FE and neighboring FEs, formally speaking the capacity for introspection and the ability to recognize oneself as an individual separate from the environment and other individuals, via its monitor and communicator. The analysis of monitor data within the evaluator followed by the deduction of actions on the local FE is performed by means of reinforcement-based machine learning classifier tables (LCTs). At this point it is important to note that these local actions may represent parts of the desired means to counteract at-runtime application interference, environmental dynamics and model uncertainties on processing resources. Generally speaking they address an EDA problem typically found in the right branch of the V-model as part of test and verification steps (cp. Fig. 2). Partial means because the ASoC FE-AE all-hardware control loop only strives to tackle control tasks that require ultimately short, i.e., few CPU clock cycles, loop latencies. At-runtime actions that are less timing critical will be accomplished by OS/Middleware or application layer techniques of the anticipated IPF architecture. An example of which, the CPSoC system, will be presented in Section IV-D2.

The two application scenarios described in the sequel will use the same three-core LEON3 Open RISC prototype with identical objective and reward functions (to demonstrate generic applicability). In the video processing application the frame rate of the I/O interface was picked such that one core can merely handle the entire workload if it operates at maximum frequency. When all three cores are available, each can run at corresponding lower frequencies and supply voltage, resulting in a more homogeneous temperature and power density distribution across the architecture. All cores have the same frame processing program loaded in their caches.

Fig. 9 shows how individual cores adjust their operation frequency in the presence of emulated randomly intermittent core failures (a maximum of two cores has been switched off in random patterns and intervals). What at first glance looks like random fluctuations in plotted frequency and CPU utilization, exhibits a trend when more closely examined. The longer the intermittent core switch off/on lasts, the less do individual cores extend the frequency range to the ultimate limits for operation with only one, respectively three, core(s) operational. LCT fitness evaluation assigned rules which more gradually follow the new system operation points over more aggressive rules. Resulting consequence at system level is that fewer frames get lost through "trapping" the multicore in operating conditions not adequate for the workload.

In the IP packet forwarding application, packet forwarding is partitioned into five tasks which can each be mapped to any core. Upon system initialization, all tasks are assigned to one and the same core (see right side of Fig. 10). The core that initially was assigned all tasks immediately runs into saturation while the other two cores are idling. The overloaded core starts issuing task migration requests to the RTE.

Task migration is accomplished by eliminating one task in the scheduling ring of one core and enabling scheduling on another core. No code copy operations are necessary, as the core-local memories (caches) host copies of all tasks. Over time we see tasks migrating among all cores with corresponding frequency adjustments to accomplish the corresponding workload. The system settles in a quasi

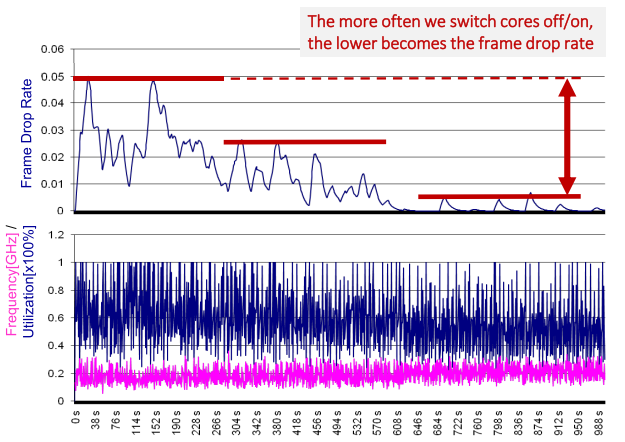


Fig. 9. *Video processing with intermittent core failures.*

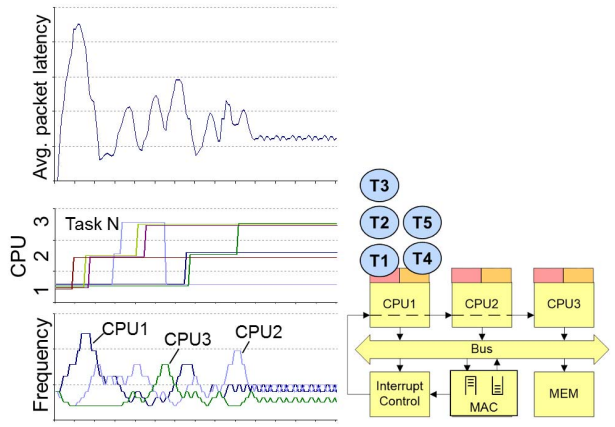


Fig. 10. *IP packet processing with automated task partitioning.*

static task partition (see left side of Fig. 10). The multicore figured out by itself how to map tasks such that no further improvement in the objective function is found. Or, expressing it differently, the self-aware multicore processor did an automated online task partitioning on parallel resources, a problem which is known to be NP hard. This time, the system level benefit expresses itself as a minimization of the overall packet processing latency. When looking carefully, we see that the system saturates in a local optimum as an even lower packet latency was experienced earlier during the partitioning process. Hence, there is no guarantee that the described approach finds the optimal solution, but a solution close to it.

We want to finish the discussion of machine learning-based self-organization of MPSoCs by discussing an important related property known as emergence. Emergent behavior occurs when system constituents (e.g., CPU cores) perform local actions (adjusting their operation parameters) which influence global system behavior through hidden causalities between the various local actions [62]. Or in short, local actions lead to global effects. Thus, emergent behavior complies with the definition of *self-organization*, which describes a process where some form of overall order or coordination arises out of the local interactions between smaller component parts of an initially disordered system. In this sense, self-awareness, based on which local actions are triggered, is a necessary precondition for self-organization and self-adaptation [63]. The phenomenon of emergent behavior can represent both a huge potential as well as a serious threat for system control.

The huge potential is that complex system-level behaviors (e.g., multicore failure resilience control or task partitioning onto parallel resources) typically result from few, fairly simple local actions (e.g., increase/decrease frequency, issue migration request for random task). Little program space, rule table size and relatively low compute performance/finite state machine control are necessary to stimulate behaviors that otherwise would be way more complex, e.g., when specified in a declarative programming language. There are several examples where the exploitation of emergent behavior can be found in nature to control complex organisms or organizations such as fish schools, flocks of birds, ant colonies, etc. In nature, evolutionary selection by the law of "survival of the fittest" did let the species with the "right" local rules survive. Organizations or organisms with "wrong" rules are extinguished, which brings us to the downside or threat of emergence. Emergent behavior may result in chaotic, instable control. There is a good base of scientific investigations that address the differentiation between constructive or controllable emergence [64], [65] and how to build trustworthy systems that inherit emergent control OC trust.[1] In technical systems, time-to-market does not allow for many-generation evolutionary selection, nor does it allow for oscillating or unstable behavior as a consequence of emergence. In technical systems, machine learning adopts the role of evolution and rapidly explores rules, separating them by fitness values. In the end, few, effective LCT rules are sufficient to "personalize and customize" a generic multicore processor for optimized usage in different application domains.

2) Cyber–Physical System-on-Chip: CPSoC [66], [67] is a smart embedded system paradigm that combines a sensor-actuator-rich self-aware computing-communication-control (C3) centric paradigm with an adaptive and reflective middleware (a flexible hardware-software stack and interface between the application and OS layer) to control the manifestations of computations (e.g., aging, overheating, parameter variability etc.) on the physical characteristics of the chip itself and the outside interacting environment. Inspired by the adaptive and learning abilities of autonomous computing [8] and C3 paradigm of CPSs [68], CPSoC provides a computing framework that assures the dependability of the cyber/information processing (i.e., the cyber aspects such as integrity, correctness, accuracy, timing, reliability and security) while simultaneously addressing the physical manifestations (in performance, power, thermal, aging, wear-out, material degradation, and reliability and dependability) of the information processing on the underlying computing platform. CPSoC aims to coalesce these two traditionally disjoint aspects/abstractions of the cyber/information world and the underlying physical computing worlds into a unified abstraction of computing by using cross-layer virtual/physical sensing and actuation, forming an ideal platform for IPFs.

The CPSoC architecture consists of a combination of sensor-actuator-rich computation platform supported by adaptive NoCs [communication NoC (cNoC) and sensor NoC (sNoC)], introspective sentient units, and an adaptive and reflective middleware to manage and control both the cyber/information and physical environment and characteristics of the chip [66], [67]. The CPSoC architecture is broadly divided into several layers of abstraction. Unlike

[1] http://gepris.dfg.de/gepris/projekt/66598707

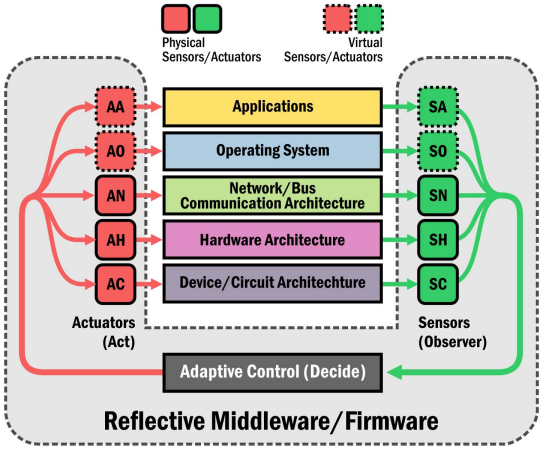


Fig. 11. ***Cross-layer virtual sensing and actuation at different layers of CPSoC.***

traditional MPSoC, each layer of the CPSoC can be made self-aware and adaptive, by a combination of software and physical sensors and actuators as shown in Fig. 11. These layer specific feedback loops are integrated into a flexible stack which can be implemented either as firmware or middleware. The CPSoC framework supports three key ideas. 1) cross-layer virtual and physical sensing and actuation—CPSoCs are sensor-actuator-rich MPSoCs that include several on-chip physical sensors (e.g., aging, oxide breakdown, leakage, reliability, temperature) on the lower three layers as shown by the on-chip-sensing-and-actuation (OCSA) block in Fig. 12. On the other hand, virtual sensing is a physical-sensorless sensing of immeasurable parameters using computation [69]. Similarly, virtual actuations (e.g., application duty cycling, algorithmic choice, checkpointing) are software/hardware interventions that can predictively influence system design objectives. Virtual actuation can be combined with physical actuation mechanisms commonly adopted in modern chips [66], [67]. 2) simple and self-aware adaptations: Two key attributes of the self-aware CPSoC are adaptation of each layer and multiple cooperative observe–decide–act loops. As an example, the unification of an adaptive computing platform (with combined dynamic voltage and frequency scaling (DVFS), adaptive body bias (ABB), and other actuation means) along with a bandwidth adaptive NoC offers extra dimensions of control and solutions in comparison to traditional MPSoC architecture. 3) Predictive models and online learning: Predictive modeling and online learning abilities enhance self-modeling abilities in the CPSoC paradigm. The system behavior and states can be built using online or offline linear or nonlinear models in time or frequency domains [19]. CPSoC's predictive and learning abilities improve autonomy for managing system resources and assisting proactive resource utilization [66], [67].

3) Non-Uniform Verification Architecture (NUVA): NUVA [70] is a scalable distributed monitoring architecture that enables low-overhead monitoring for parametric specifications over multiprocessor systems in the form of parameterized finite-state automata, intended for embedded and general-purpose multiprocessors deployed within CPSs and IoTs. The core of NUVA is a coherent distributed automata transactional memory that efficiently maintains states of a dynamic population of automata checkers organized into a rooted dynamic directed acyclic graph concurrently shared among all processor nodes. NUVA comprises five ingredients: 1) a specification language based on self-replicating finite automata (SR-DFAs) introduced in [70]; 2) a naturally distributed representation [70], [71] of RV checkers and auxiliary information; 3) decision algorithms; 4) a low-overhead RV architecture; and 5) a specification mining tool called ParaMiner. The distributed RV architecture is the centerpiece, and namesake, of NUVA. It is intrinsically distributed and scalable, minimizes conflicts among concurrent RV transactions over shared RV data, and is loosely coupled and minimally invasive to current CPU architectures. Many RV frameworks have been proposed with varying degrees of success. NUVA is the first attempt at solving two open problems. 1) Most RV frameworks lacked a naturally distributed structure that scales RV to arbitrarily large multiprocessor systems. 2) Pure software RV frameworks sustain substantial performance and power overheads. As shown in Fig. 13, NUVA tackles both problems through architectural support for RV of parametric specifications. A vital quality of NUVA is that parametric (or data-carrying) event streams are so general as to possibly stand for many different aspects of program executions, user behavior, environment conditions (battery, temperature, GPS, sensory data, text, video, etc.). NUVA is able to handle all of these event streams in the same specification.

Specification Mining: In general, automated software verification decides correctness of a design against a formal specification. Unfortunately, formal specifications, like

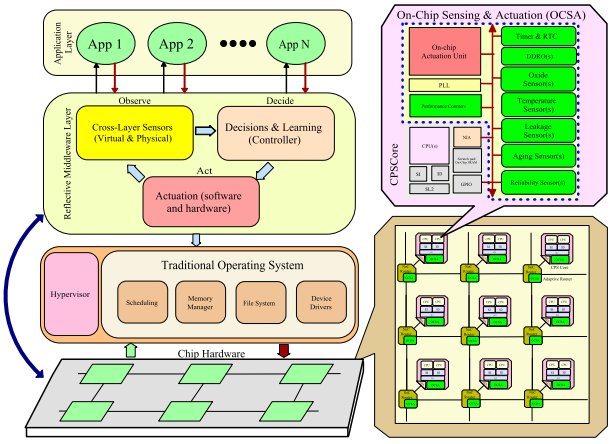


Fig. 12. ***CPSoC architecture with adaptive Core, NoC, and the observe–decide–act loop as adaptive, reflective middleware.***

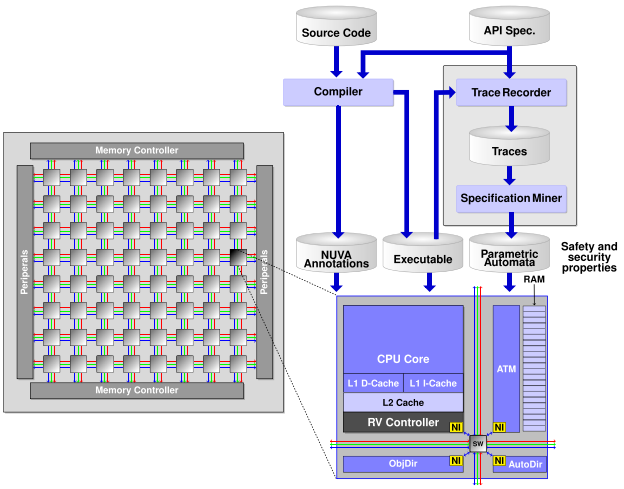


Fig. 13. ***Hardware-assisted cross-layer, specification-based monitoring infrastructure.***

SR-DFAs, are notoriously hard to formulate and maintain for evolving complex distributed systems, especially at the level of precision mandated by such applications as host-based intrusion detection. This difficulty hinders formal specifications from coping with agile, fast-changing computing environments. Additionally, specifications of existing software systems are needed to verify (or secure) new systems built on top of them. Therefore, specification mining [72] emerged as an automated technique used to discover formal specifications of computing systems from samples of their executions. Inferred properties can take many forms, such as value invariants, finite-state machines, temporal properties, or sequence diagrams. The NUVA framework includes ParaMiner, a tool that discovers software properties (in the form of SR-DFAs [70]) of arbitrary, tunable complexity and precision from large execution traces. ParaMiner relies on a novel specification mining technique that is the first to introduce and use parametric multiple sequence alignment (pMSA) that extends classical MSA [73] to handle parameterized alphabet. In [74], we presented sound theoretical underpinnings of using MSA as a language learning tool for the case of classical finite automata. Using MSA, ParaMiner can reconstruct properties with abstract state spaces which do not merely duplicate the hidden program state space and whose sizes are dictated solely by the complexity of observed execution traces.

E. Exemplar Design Driver: Autonomous Mobile System

In the following, we use a smart phone use case to outline how platform control and management of the IP factory concept can be applied to complex systems. For smart phones, the IPF concept spans through multiple abstraction levels of the device, including hardware, OS and application layers (cp. Fig. 7) and interacts within and outside itself as part of an ecosystem comprising the device, the mobile infrastructure and the cloud. Traditionally, power consumption is a primary goal but other considerations are also important such as size, cost, security and privacy. The latter two become especially important as much of our information is now accessed through these devices. As mobile systems become increasingly powerful and diverse in their scope, the ability to foretell all the possible scenarios they will go through in their lifetime quickly diminishes, which could lead to problems of massive nature. Nowhere is that more apparent than in the recent recall of Samsung's Note 7 which is estimated to have cost Samsung close to $17 billion in lost revenue. As smart phones become more complex and empowered with additional capabilities, the IP factory model becomes indispensable in making these devices more self-aware, and in doing so, making them more energy efficient but at the same time increasing their resiliency to different "anomalies" such as security (intrusion), aging, etc. The key issues in this self-awareness is to be able to encompass, manage and evolve the interactions between layers of abstraction as well as the interactions within one layer.

For illustration, consider a smart phone SoC where voltage overscaling (VOS) is applied at the *hardware* layer to control power consumption. The goal is to allow the hardware to contribute to the noise floor of the system in a *controlled* manner as long as that leads to minimizing the cost in terms of power and deviation from QoS metrics. For example, if VOS is used at the physical layer (for a WCDMA radio link), errors gradually start to appear in the hardware. These errors will then propagate through the system and manifest themselves at the application layer.

At the *platform* level, VOS can be applied to both the modem and the application processor (an H.264 video decoder in this case) separately (cases A and B) or jointly (case C). Fig. 14 shows the results of these three policies, concluding that although more involved, a joint policy can achieve better tradeoff between power and video

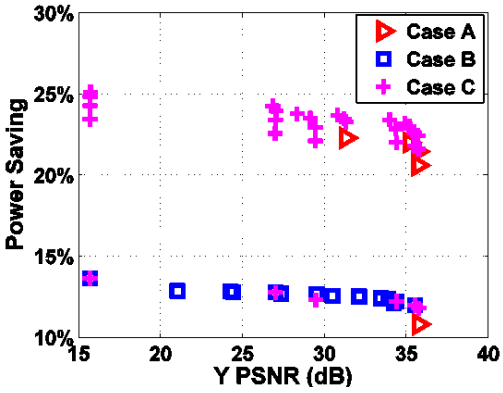


Fig. 14. ***Combined modem and application processor design space exploration.***

quality. The situation is not as simple as one might imagine, though. There exists a complex relationship between power, performance (i.e., frequency) and reliability (i.e., probability of errors) factoring in temperature in a SoC Upscaling voltage achieves higher noise immunity and higher performance, but results in increased power consumption, which raises the chip temperature, thereby increasing leakage power, which further increases chip temperature until the package dissipation limit is reached. This increased temperature results in decreased reliability and potentially offsetting expected gains in resilience originally sought by increasing voltage.

Tradeoffs at different layers can be exploited to optimize power and reliability. At the *modem* layer, it is possible to use a hybrid adaptive modulation control scheme to tradeoff modulation (i.e., bitrate) versus power at different signal-to-noise ratio (SNR) levels. At the *network* layer, we will assume that selective protection and UDP-Lite implementations are available. At the *application* layer we can use probability-based power aware intrarefresh (PBPAIR) [75]. The PBPAIR scheme inserts intracoding (i.e., coding without reference to any other frame) to enhance the robustness of the encoded bitstream at the cost of compression efficiency. Another technique at the *application ecosystem* layer is video transcoding which requires a cooperation with the remote video server in the cloud to generate an encoded video specifically tailored to each hardware's defect map. By controlling the macroblock sizes, it is possible to limit the error propagation, thus enabling higher peak SNR at reduced power levels.

In the IP factory context, a coordination across all layers and within each layer is necessary. Developing such a capability requires sensing and actuation across both the physical and cyber domains at the interface level, and self-awareness within the system. In this illustrative example a particularly challenging task is the development of an intelligent cross-layer power management policy that can adaptively tune itself based on the current state of the system. For instance, given a system running at a certain level of performance, power and reliability, the policy will need to react to changes in the environment (e.g., channel conditions, noise, user input, temperature, etc.). There may be a number of alternative actions that can be explored at different abstraction levels. At the physical layer: V_{dd} scaling and/or V_{sb} (substrate voltage) scaling, which may increase or decrease power with side effects on performance, temperature. At the architectural layer: processor frequency scaling, migration of jobs between processors (e.g., to eliminate hot spots on the chip) or throttling communication between processors to reduce temperature. This can, e.g., be achieved systematically by dynamic resource management [76], [77] and at the application layer by changing the mode of operation, for example reducing the frame rate in video or reducing the resolution by dropping the enhancement layers. Additionally, one may choose to apply any of these techniques to one application or to multiple applications simultaneously as illustrated in the video-over-wireless case study. The big question is: which of these alternatives should be employed for a given set of circumstances? The main premise of this thrust is to practically demonstrate that instead of having separate, narrowly scoped policies, much more benefit can be reaped by approaching the problem comprehensively. The cost, of course, is an increase in the policy complexity, but this is something that can be mitigated by the fact that today's chips have significantly more space and computing power than before. With that premise in mind, enabling the horizontal (i.e., within a layer) and vertical (cross layer) coordination requires the chip itself to be cognizant of much more than a limited set of parameters or facets. This broad-based perception can be thought of as platform self-awareness. The self-aware IPF paradigm provides a conceptual framework under which these challenges of cross layer and within layer cognizance can be explored and addressed.

F. IPF in Critical Systems

Up to here, the IPF approach lacks the rigorous handling of critical requirements. In fact, at the current state there is no enforcement of timing or safety constraints in this. Unlike the CCC use case, where possible platform changes are included in safety and availability margins reflecting the established approach to critical systems design, IPF handles significant long-term platform changes and, therefore, must actively control platform components and their parameters to safeguard critical requirements. Possible methods include constraint-driven dynamic resource management [76], [77], which gives preference to critical tasks when needed, and fast error detection and recovery mechanisms when errors have occurred [78], [79]. Safety standards as discussed above, however, not only require functional guarantees but also preservation of safety margins to cope with short term error effects across a system stack like in Fig. 7. Such "margin guarantees" could possibly be provided by automated methods as developed for the CCC use case, but assessment of margins requires model accuracy which is unknown beyond the current operating point of a critical system. This limitation is a general challenge of current approaches to self-awareness and must be overcome if IPF methods shall be used for critical functions, as well. Here, system identification could be applied as introduced previously; by controlled operating parameter change (voltage, load, etc.), system behavior could be explored in the neighborhood exploiting (but not impacting) the available margin. Once the current model accuracy has been investigated this way, similar monitoring mechanisms as in the CCC use case could be applied. Both cross-layer margins and models with a wider range of guaranteed accuracy remain relevant but open research issues.

V. EDA CHALLENGES

The two use cases elaborated in the preceding sections are, to a large extent, complementary. Using different

architectures and methods they have illustrated the opportunities arising from self-awareness in CPSs. But the use cases also show some key challenges that entail new EDA problems.

A. Model Validation

While the current design process assumes correct models or, at least, the capability to validate model correctness in the lab, a self-aware system must cope with model uncertainty and incorrectness. In the IPF case, we assume that system changes lead to model deviation and hold the system itself responsible for model adaptation. IPF uses system identification techniques but such techniques are limited to few parameters that are controllable and observable, and they require sufficient system margin to be applicable. For larger systems, we need research how to partition a system for identification. Partitioning should eventually be executed in the field because the system structure might change over time. In the CCC case, model correctness stays a designer responsibility while the self-aware system is responsible for model integration. Initially trusting the designer provided model, the self-aware system focuses on dependencies and formal analysis coupled with self-configuration. While in IPF, model uncertainty is the rule, in CCC it is considered an exception requiring protective mechanisms. Both methods of IPF and CCC can be combined opening a large new area of self-modeling and model prediction for large systems with online model validation.

B. Designing the Self-Aware Design Process

With self-awareness moving part of the design process to the field, the designer develops a system that continues designing itself. Decisions at design time including objectives and constraints have an indirect effect on the resulting behavior and structure. Controlling such a process requires a new form of system specification that does not define a single behavior, but constrains and guides the expected system behavior. Specifying and controlling such a "design corridor" is another new EDA problem of high importance not only to self-aware systems but to autonomous systems in general.

C. Critical Systems Design Automation

Currently, critical systems design is highly user driven leaving many opportunities for EDA research. Definition of safety cases, application of error models, dependency determination and error propagation are still genuine engineering tasks. While the definition of safety cases requires profound knowledge of application functions and their effects, dependability analysis uses more systematic formal methods, such as FMEA, which could be automated and executed in the field. One of the main automation challenges is the determination of relevant dependencies in a complex system as demonstrated in the CCC use case. Automation of this design step would be helpful in a traditional design process, but is an essential requirement for changing and autonomous systems. While self-awareness alone does not solve that challenge, its cross-layer modeling capabilities provide a good basis for formal dependability analysis. Such dependability analysis could, then, be used for self-configuration to improve or restore guaranteed margins. Here, integration in the CCC use case is closer to a solution, because the IPF approach faces the additional challenge of system self-identification as explained above. System self-identification for dependability analysis is an important research topic in itself, because it could be used for early detection of latent degradation or defects in general. Thus, system self-identification could become the foundation of a new generation of monitoring and self-diagnosis functions greatly improving CPS dependability.

D. Software Architectures and Efficiency

The CCC use case demonstrates the need for new software architectures and runtime environments. Embedding the model domain, protocols for the software "self-management", the generation of monitors, CCC gives implementation examples from a wide range of possible solutions that are largely unexplored. While dependability and IT security can profit from additional information and methods provided by such a self-awareness infrastructure, this infrastructure itself becomes a primary functional safety concern and arises as an object of IT security attacks. CCC could only provide initial results which fit the use case, but more research is necessary for larger and open CPSs. Algorithm efficiency and frequency of execution determine the required runtime and energy overhead for self-awareness. This is another research field with impact on the practical feasibility and acceptance of self-aware systems.

E. Self-Awareness and Machine Learning

Self-awareness can favorably be used for machine learning in many forms. It can improve predictability and dependability by monitoring input data uncertainty and controlling the effects of learned behavior by predicting the consequences of actions. The self-modeling and prediction infrastructure could even be used to support training ("self-reflection"). There seem to be many opportunities in combining self-awareness and machine learning.

F. Self-Awareness, Emergence, and Scalable Control

Coupling several self-aware systems for cooperative control has shown to be an effective mechanism for distributed control, at least on a smaller scale [49], [80]. In the IPF use case, many layers of different types must be coupled vertically, while in the CCC example, many participants interact on the same level (horizontally). Large interacting

self-aware systems show emergent system behavior which can be intended to achieve new functionality but must be controlled to stay within the intended design corridor. Hence, heterogeneity, emergence, and scalability will arise as general challenges for self-aware systems collaboration requiring closer investigation of the resulting behavior.

VI. CONCLUSION

Managing change and evolution is a major challenge for future CPSs, particularly for the methods used to design them. Self-awareness is the ability (of a computing system) to recognize its own state, possible actions and the result of these actions on itself and its environment.

The two complementary use cases in this paper employ different architectures and methods and have illustrated the opportunities arising from self-awareness in CPSs. Both share the principles of self-modeling, self-configuration, and monitoring. They maintain a continuously updated image of the system state, but differ in the objectives and the concrete approach to meet the design goals. While the IPF use case relies on incremental changes and feedback control to track the evolution of system parameters and optimize system properties, CCC follows an analytical approach with contracting, formal system analysis, self-configuration and contract enforcement. While IPF targets flexible platform adaptation for changing CPS requirements, CCC mainly addresses critical CPSs using a rigorous process following the principles of current critical systems design. Both approaches could be combined. They introduce new methods that effectively extend design processes from the lab to the field adding new system capabilities that are not achievable in current lab-based design processes. This feature extension opens a new dimension of systematic CPS control and management thereby heavily relying on design automation. This results in a variety of opportunities for EDA research as outlined in the paper. Notably, two major concerns of future CPSs, control of machine learning and improvement of systems security, can profit from this research. ■

Acknowledgments

The authors would like to thank J. Zeppenfeld and A. Bernauer for their valuable contribution to the ASoC project, and S. Sarma for his valuable contribution to the CPSoC project. M. Möstl, J. Schlatow, and R. Ernst were part of the CCC project. N. Dutt, A. Nassar, A. Rahmaniy, F. J. Kurdahi, T. Wild, A. Sadighi, and A. Herkersdorf were part of the IPF project.

REFERENCES

[1] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, Standard IEC 61508, 2nd ed., Int. Electrotechnical Commission, Apr. 2010.

[2] *Road Vehicles—Functional Safety*, Standard ISO 26262, International Organization for Standardization, Apr. 2011.

[3] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Washington, DC, USA, Dec. 2011.

[4] R. Ernst and M. Di Natale, "Mixed criticality systems—History of misconceptions?" *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, Oct. 2016.

[5] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer, Apr. 2011.

[6] *ARINC Specification 653 Parts 1-2*, ARINC, Annapolis, MD, USA, Aug. 2015.

[7] *Analysis Techniques for System Reliability—Procedure for Failure Mode and Effects Analysis (FMEA)*, Standard IEC 60812, 2nd ed., Int. Electrotechnical Commission, Jan. 2006.

[8] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[9] P. Padala, "Adaptive control of virtualized resources in utility computing environments," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst. (EuroSys)*, 2007, pp. 289–302.

[10] R. Isermann, *Fault-Diagnosis Systems: An Introduction From Fault Detection to Fault Tolerance*. Springer, Jan. 2006.

[11] A. Morin, "Levels of consciousness and self-awareness: A comparison and integration of various neurocognitive views," *Consciousness Cogn.*, vol. 15, no. 2, pp. 358–371, Jun. 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1053810005001224

[12] P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, *Self-Aware Computing Systems—An Engineering Approach*. Springer, 2016.

[13] P. R. Lewis, "A survey of self-awareness and its application in computing systems," in *Proc. Conf. Self-Adapt. Self-Org. Syst. Workshops (SASOW)*, Oct. 2011, pp. 102–107.

[14] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, "Adaptivity and self-organization in organic computing systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 5, no. 3, p. 10, Sep. 2010.

[15] H. Hoffmann, "JouleGuard: Energy guarantees for approximate applications," in *Proc. 25th Symp. Oper. Syst. Princ. (SOSP)*, 2015, pp. 198–214.

[16] H. Hoffmann, "CoAdapt: Predictable behavior for accuracy-aware applications running on power-aware systems," in *Proc. 26th Euromicro Conf. Real-Time Syst.*, Jul. 2014, pp. 223–232.

[17] H. Giese, T. Vogel, A. Diaconescu, S. Götz, and S. Kounev, "Architectural concepts for self-aware computing systems," in *Self-Aware Computing Systems*. Cham, Switzerland: Springer, 2017.

[18] A. Jantsch and K. Tammemäe, "A framework of awareness for artificial subjects," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synthesis (CODES)*, Oct. 2014, pp. 1–3.

[19] L. Ljung, "System identification," in *Signal Analysis and Prediction* (Applied and Numerical Harmonic Analysis), A. Procházka, J. Uhlíř, P. W. J. Rayner, and N. G. Kingsbury, Eds. Boston, MA, USA: Birkhäuser, 1998.

[20] M. Neukirchner, K. Lampka, S. Quinton, and R. Ernst, "Multi-mode monitoring for mixed-criticality real-time systems," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, Sep. 2013, pp. 1–10.

[21] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, "Sandboxing controllers for cyber-physical systems," in *Proc. IEEE/ACM 2nd Int. Conf. Cyber-Phys. Syst. (ICCPS)*, Apr. 2011, pp. 3.1–3.12.

[22] *Fault Tree Analysis (FTA)*, document IEC 61025, 2nd ed., Int. Electrotechnical Commission, Dec. 2006.

[23] M. Moestl and R. Ernst, "Cross-layer dependency analysis for safety-critical systems design," in *Proc. Archit. Comput. Syst.*, Mar. 2015, pp. 1–7.

[24] J. Schlatow, M. Moestl, and R. Ernst, "An extensible autonomous reconfiguration framework for complex component-based embedded systems," in *Proc. Int. Conf. Autono. Comput. (ICAC)*, Jul. 2015, pp. 239–242.

[25] J. Schlatow, M. Nolte, M. Möstl, I. Jatzkowski, R. Ernst, and M. Maurer, "Towards model-based integration of component-based automotive software systems," in *Proc. 43rd Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Beijing, China, Oct. 2017, pp. 8425–8432.

[26] M. Hamad, M. Nolte, and V. Prevelakis, "Towards comprehensive threat modeling for vehicles," in *Proc. Workshop Secur. Dependab. Crit. Embedded Real-Time Syst. (CERTS)*, 2016, pp. 31–36.

[27] W. Xu, M. Wegner, L. Wolf, and R. Kapitza, "Byzantine agreement service for cooperative wireless embedded systems," in *Proc. 3rd Int. Workshop Saf. Secur. Intell. Vehicles (SSIV)*, Denver, CO, USA, Jun. 2017, pp. 10–15.

[28] A. Dörflinger, B. Fiethe, H. Michalik, P. Keldenich, and C. Scheffer, "Resource-efficient dynamic partial reconfiguration on FPGAs for space instruments," in *Proc. 11th NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Pasadena, CA, USA, Jul. 2017, pp. 24–31.

[29] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *Proc. 22nd IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2016, pp. 1–10.

[30] J. Schlatow and R. Ernst, "Response-time analysis for task chains with complex precedence and blocking relations," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, Seoul, South Korea, Oct. 2017, p. 172.

[31] W. Xu, Z. A. H. Hammadeh, A. Kröller, R. Ernst, and S. Quinton, "Improved deadline miss models for real-time systems using typical worst-case analysis," in *Proc. 27th Euromicro Conf. Real-Time Syst.*, Jul. 2015, pp. 247–256.

[32] M. Moestl and R. Ernst, "Handling complex dependencies in system design mischa," in *Proc. Design, Automat. Test Eur. (DATE)*, 2016, pp. 1120–1123.

[33] H. Härtig, "Security architectures revisited," in *Proc. 10th ACM SIGOPS Eur. Workshop*, 2002, pp. 16–23.

[34] A. J. G. Klein, "seL4: Formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, p. 107–115, Jun. 2010.

[35] R. Buerki and A.-K. Rueegsegger, "Muen—An x86/64 separation kernel for high assurance," Tech. Rep., Aug. 2013.

[36] P. J. Prisaznuk, "ARINC 653 role in integrated

modular avionics (IMA)," in *Proc. IEEE/AIAA 27th Digit. Avionics Syst. Conf.*, Oct. 2008, pp. 1–3.
[37] *PikeOS Hypervisor*. [Online]. Available: https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/
[38] M. Beckert, K. B. Gemlau, and R. Ernst, "Exploiting sporadic servers to provide budget scheduling for ARINC653 based real-time virtualization environments," in *Proc. Desing Automat. Test Eur. (DATE)*, Lausanne, Switzerland, Mar. 2017, pp. 870–875.
[39] A. Lyons and G. Heiser, "Mixed-criticality support in a high-assurance, general-purpose microkernel," in *Proc. Workshop Mixed Criticality Syst.*, Dec. 2014, pp. 9–14.
[40] M. Stein, "A kernel in a library: Genode's Custom kernel approach," in *Proc. FOSDEM*, Brussels, Belgium, Feb. 2017. [Online]. Available: https://fosdem.org/2017/schedule/event/microkernel_kernel_library/
[41] N. Feske, "GENODE. Operating system framework 18.05. Foundations," Tech. Rep., 2018. [Online]. Available: http://genode.org/documentation/genode-foundations-18-05.pdf
[42] J. Song, Q. Wang, and G. Parmer, "The state of composite," in *Proc. Workshop Oper. Syst. Platforms Embedded Real-Time Appl.*, 2013.
[43] M. Fernandez, G. Klein, I. Kuz, and T. Murray, "CAmkES formalisation of a component platform," NICTA UNSW, Sydney, NSW, Australia, Australia, Tech. Rep., Nov. 2013.
[44] M. Hamad, J. Schlatow, V. Prevelakis, and R. Ernst, "A communication framework for distributed access control in microkernel-based systems," in *Proc. Oper. Syst. Platforms Embedded Real-Time Appl.*, 2016, pp. 11–16.
[45] V. Prevelakis and M. Hamad, "A policy-based communications architecture for vehicles," in *Proc. Int. Conf. Inf. Syst. Secur. Privacy (ICISSP)*, Feb. 2015, pp. 155–162.
[46] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in C^3," in *Proc. IEEE 34th Real-Time Syst. Symp.*, Dec. 2013, pp. 21–32.
[47] J. Song, G. Bloom, and G. Parmer, "SuperGlue: IDL-based, system-level fault tolerance for embedded systems," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Networks (DSN)*, Jun. 2016, pp. 227–238.
[48] L. Thiele, F. Sutton, R. Jacob, R. Lim, R. Da Forno, and J. Beutel, "On platforms for CPS-adaptive, predictable and efficient," in *Proc. Int. Symp. Rapid Syst. Prototyping (RSP)*, Oct. 2016, pp. 1–3.
[49] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A general and extensible framework for self-aware computing," Tech. Rep., 2011.
[50] H. Giese, *State of the Art in Architectures for Self-Aware Computing Systems*. Cham, Switzerland: Springer, 2017.
[51] A. Bennaceur, "Mechanisms for leveraging models at runtime in self-adaptive software," in *Models@Run.Time* (Lecture Notes in Computer Science), N. Bencomo, R. France, B. H. C. Cheng, and U. Assmann, Eds. Springer, 2014.
[52] B. H. C. Cheng, "Using models at runtime to address assurance for self-adaptive systems," in *Models@Run.Time* (Lecture Notes in Computer Science), N. Bencomo, R. France, B. H. C. Cheng, and U. Assmann, Eds. Springer, 2014.
[53] J. Song, G. Fry, C. Wu, and G. Parmer, "CAML: Machine learning-based predictable, system-level anomaly detection," in *Proc. Workshop Secur. Dependability Crit. Embedded Real-Time Syst.*, 2016, pp. 12–18.
[54] (2001–2017). *QNX Neutrino RTOS*. [Online]. Available: http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html
[55] T. Stolte, G. Bagschik, A. Reschka, and M. Maurer, "Hazard analysis and risk assessment for an automated unmanned protective vehicle," *CoRR*, vol. abs/1704.06140, Apr. 2017.
[56] P. Axer, D. Thiele, and R. Ernst, "Formal timing analysis of automatic repeat request for switched real-time networks," in *Proc. 9th IEEE Int. Symp. Ind. Embedded Syst. (SIES)*, Jun. 2014, pp. 78–87.
[57] L. Wanner, "NSF expedition on variability-aware software: Recent results and contributions," *Inf. Technol.*, vol. 57, no. 3, pp. 181–198, 2015. [Online]. Available: https://www.degruyter.com/view/j/itit.2015.57.issue-3/itit-2014-1085/itit-2014-1085.xml
[58] J. Henkel, "Design and architectures for dependable embedded systems," in *Proc. 9th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synthesis (CODES+ISSS)*, Oct. 2011, pp. 69–78.
[59] A. Bernauer, O. Bringmann, and W. Rosenstiel, "Generic self-adaptation to reduce design effort for system-on-chip," in *Proc. 3rd IEEE Int. Conf. Self-Adapt. Self-Organizing Syst.*, Sep. 2009, pp. 126–135.
[60] J. Zeppenfeld and A. Herkersdorf, "Applying autonomic principles for workload management in multi-core systems on chip," in *Proc. 8th ACM Int. Conf. Auton. Comput.*, 2011, pp. 3–10.
[61] M. V. Butz, *Rule-Based Evolutionary Online Learning Systems—A Principled Approach to LCS Analysis and Design*. Springer, 2006.
[62] J. Fromm, *The Emergence of Complexity*. Kassel Univ. Press, 2004.
[63] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, p. 14, May 2009.
[64] C. Müller-Schloer, H. Schmeck, and T. Ungerer, *Organic Computers—A Paradigm Shift for Complex Systems*. Basel, Switzerland: Springer, 2011.
[65] D. Fisch, M. Janicke, B. Sick, and C. Müller-Schloer, "Quantitative emergence—A refined approach based on divergence measures," in *Proc. 4th IEEE Int. Conf. Self-Adapt. Self-Organizing Syst.*, Sep. 2010, pp. 94–103.
[66] S. Sarma, N. Dutt, P. Gupta, N. Venkatasubramanian, and A. Nicolau, "CyberPhysical-System-On-Chip (CPSoC): A self-aware MPSoC paradigm with cross-layer virtual sensing and actuation," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2015, pp. 625–628.
[67] S. Sarma, N. Dutt, N. Venkatasubramanian, A. Nicolau, and P. Gupta, "Cyber Physical-System-on-Chip (CPSoC): Sensor-actuator rich self-aware computational platform," Univ. California, Irvine, CA, USA, Tech. Rep., 2013.
[68] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. 11th IEEE Int. Symp. Object Compon.-Oriented Real-Time Distrib. Comput. (ISORC)*, May 2008, pp. 363–369.
[69] S. Sarma, N. Dutt, and N. Venkatasubramanian, "Cross-layer virtual observers for embedded multiprocessor system-on-chip (MPSoC)," in *Proc. 11th Int. Workshop Adapt. Reflective Middleware*, 2012, p. 4.
[70] A. Nassar, F. J. Kurdahi, and W. Elsharkasy, "NUVA: Architectural support for runtime verification of parametric specifications over multicores," in *Proc. Int. Conf. Compil., Archit. Synthesis Embedded Syst.*, 2015, pp. 137–146.
[71] A. Nassar and F. J. Kurdahi, "Lattice-based Boolean diagrams," in *Proc. 21st Asia South Pacific Design Automat. Conf. (ASP-DAC)*, 2016, pp. 468–473.
[72] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2002, pp. 4–16.
[73] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge, U.K.: Cambridge Univ. Press, 1998.
[74] A. Nassar, F. J. Kurdahi, and S. R. Zantout, "Topaz: Mining high-level safety properties from logic simulation traces," in *Proc. Conf. Design, Automat. Test Eur. (DATE)*, Mar. 2016, pp. 1473–1476.
[75] M. Kim, H. Oh, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "PBPAIR: Probability based power aware intra refresh—A new energy-efficient error resilient coding scheme," Center Embedded Comput. Syst., Univ. California, Irvine, CA, USA, Tech. Rep., 2005.
[76] A. Kostrzewa, S. Saidi, and R. Ernst, "Dynamic control for mixed-critical networks-on-chip," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2015, pp. 317–326.
[77] A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst, "Dynamic admission control for real-time networks-on-chips," in *Proc. 21st Asia South Pacific Design Automat. Conf. (ASP-DAC)*, Jan. 2016, pp. 719–724.
[78] E. A. Rambo, C. Seitz, S. Saidi, and R. Ernst, "Designing networks-on-chip for high assurance real-time systems," in *Proc. IEEE 22nd Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Jan. 2017, pp. 185–194.
[79] E. A. Rambo, S. Saidi, and R. Ernst, "Providing formal latency guarantees for ARQ-based protocols in networks-on-chip," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2016, pp. 103–108.
[80] H. Hoffmann, "Self-aware computing in the Angstrom processor," in *Proc. Design Automat. Conf. (DAC)*, Jun. 2012, pp. 259–264.

ABOUT THE AUTHORS

Mischa Möstl received the B.S. and M.S. degrees in computer and communication systems engineering from the Technische Universität Braunschweig, Braunschweig, Germany, in 2011 and 2013, respectively, where he is currently working toward the Ph.D. degree at the Institute of Computer and Network Engineering (IDA) under Prof. R. Ernst and a member of the CCC project.

His research interests are in-field safety validation for systems under concurrent change and self-aware mechanisms for safety.

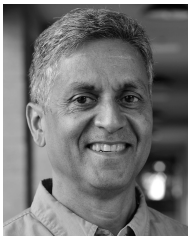
Johannes Schlatow received the M.Sc. degree in computer and communication systems engineering from the Technical University of Braunschweig, Braunschweig, Germany, in 2013, where he is currently working toward the Ph.D. degree.

He is a Researcher in the Embedded System Design Automation Group (IDA) of Prof. R. Ernst. He is working in the field of design, modeling, and analysis of component-based mixed-critical systems and a member of the CCC project.

Rolf Ernst (Fellow, IEEE) received the Diploma degree in computer science and the Dr.Ing. degree in electrical engineering from the University of Erlangen-Nuremberg, Erlangen, Germany, in 1981 and 1987, respectively.

After two years with Bell Laboratories, Allentown, PA, USA, he joined the Technische Universitaet Braunschweig, Braunschweig, Germany, as a Professor of Electrical Engineering. He chairs the Institute of Computer and Network Engineering (IDA) covering embedded systems research from computer architecture and real-time systems theory to challenging automotive, aerospace, or smart building applications. He co-ordinates the DFG Research Group Controlling Concurrent Change and is a member of the IPF project.

Prof. Ernst is a DATE Fellow. He served as an ACM SIGDA Distinguished Lecturer, and is a member of the German Academy of Science and Engineering (acatech). In 2014, he received the annual Achievement Award of the European Design Automation Association (EDAA).



Nikil D. Dutt (Fellow, IEEE) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, IL, USA, in 1989.

He is currently a Distinguished Professor of Computer Science, Cognitive Sciences, and Electrical Engineering and Computer Science at the University of California Irvine, Irvine, CA, USA. He is also a Distinguished Visiting Professor in the CSE Department, IIT Bombay, India. He is a coauthor of seven books on topics covering hardware synthesis, memory and computer architecture specification and validation, and on-chip networks. His research interests are in embedded systems, electronic design automation (EDA), computer systems architecture and software, healthcare Internet-of-Things (IoT), and brain-inspired architectures and computing.

Prof. Dutt is a Fellow of the Association for Computing Machinery (ACM) and recipient of the IFIP Silver Core Award. He received over a dozen best paper awards and nominations at premier EDA and embedded systems conferences. He has served as Editor-in-Chief of the ACM Transactions on Design Automation of Electronic Systems (TODAES) and as an Associate Editor for the ACM Transactions on Embedded Computing Systems (TECS) and the IEEE Transactions on Very Large Scale Intergation (VLSI) Systems. He has extensive service on the steering, organizing, and program committees of several premier EDA and Embedded System Design conferences and workshops, and also serves or has served on the advisory boards of ACM SIGBED, ACM SIGDA, ACM TECS, IEEE Embedded Systems Letters (ESL), and the ACM Publications Board.

Ahmed Nassar received the B.Sc. degree in electronics and communications engineering from Alexandria University, Egypt, in 2002, the M.Sc. degree in electronics engineering from Cairo University, Cairo, Egypt, in 2010, and the Ph.D. degree in electrical and computer engineering from the University of California Irvine, Irvine, CA, USA, in 2016.

He is currently with NVIDIA Corp., Santa Clara, CA, USA. His research interests lie in the area of modeling, design, and verification of cyber-physical systems in general, and in verification and safety validation of automated driving systems in recent years.

Dr. Nassar won a best paper award from ASP-DAC 2016.

Amir M. Rahmani (Senior Member, IEEE) received the M.S. degree from the Department of Electrical and Computer Engineering, University of Tehran, Tehran, Iran, in 2009, the Ph.D. degree from the Department of Information Technology, University of Turku, Finland, in 2012, and the MBA degree jointly from the Turku School of Economics and the European Institute of Innovation & Technology (EIT) ICT Labs, in 2014.

He is currently Marie Curie Global Fellow at the University of California Irvine, Irvine, CA, USA and Technical University of Vienna (TU Wien), Vienna, Austria. He is also an Adjunct Professor (Docent) in embedded parallel and distributed computing at the University of Turku. He is the author of more than 160 peer-reviewed publications. His research interests span self-aware computing, energy-efficient many-core systems, runtime resource management, healthcare Internet-of-Things, and fog/edge computing.

Dr. Rahmani has served on a large number of technical program committees of international conferences, such as DATE, VLSID, GLSVLSI, DFT, ESTIMedia, CCNC, MobiHealth, and others, and was a guest editor for special issues in journals such as the Journal of Parallel and Distributed Computing, Future Generation Computer Systems, the ACM Springer Mobile Networks and Applications (MONET) Journal, Sensors, Supercomputing, etc.

Fadi Kurdahi (Fellow, IEEE) received the Ph.D. degree from the University of Southern California, Los Angeles, CA, USA, in 1987.

Since then, he has been with the Faculty at the Department of Electrical and Computer Engineering at the University of California Irvine, Irvine, CA, USA, where he conducts research in the areas of computer-aided design and design methodology of large scale systems. Currently, he serves as the Associate Dean for Graduate and Professional Studies at the Henry Samueli School of Engineering, and the Director of the Center for Embedded and Cyber-Physical Systems (CECS), comprising world-class researchers in the general area of embedded and cyber-physical systems.

Dr. Kurdahi served on numerous editorial boards, and was Program Chair or General Chair on program committees of several workshops, symposia, and conferences in the area of computer-aided design (CAD), very large scale intergation (VLSI), and system design. He received the Best Paper Awards for the IEEE Transactions on Very Large Scale Intergation (VLSI) Systems in 2002, ISQED in 2006, and ASP-DAC in 2016, and other distinguished paper awards at DAC, EuroDAC, ASP-DAC, and ISQED. He also received the Distinguished Alumnus Award from his Alma Mater, the American University of Beirut, in 2008. He is a Fellow of the American Association for the Advancement of Science (AAAS).

Thomas Wild received the Dipl.-Ing. and Dr.-Ing. degrees from the Department of Electrical and Computer Engineering, Technical University of Munich (TUM), Munich, Germany, in 1989 and 2003, respectively.

He is a member of the scientific staff at the Chair of Integrated Systems (LIS), TUM, and is responsible for the activities in the area of multicore and network processing architectures. His current research interests comprise multiprocessor system-on-chip (MPSoC) architectures, networks-on-chip (NoC) and memory hierarchies as well as MPSoC diagnosis, system level design methodologies, and design space exploration.

Armin Sadighi received the B.S. degree in computer engineering from Amirkabir University of Technology, Tehran, Iran, in 2013, and the M.S. degree in communication electronics from the Technical University of Munich, Munich, Germany, in 2016, where he is currently working toward the Ph.D. degree in the Electrical Engineering Department.

His research interests include application-specific multiprocessor architectures, self-aware computing, and autonomous systems.

Andreas Herkersdorf (Senior Member, IEEE) received the Ph.D. degree from ETH Zurich, Zurich, Switzerland, in 1991.

He is a Professor in the Department of Electrical and Computer Engineering and also affiliated with the Department of Informatics, Technical University of Munich (TUM), Munich, Germany. Between 1988 and 2003, he was in technical and management positions with the IBM Research Laboratory, Rüschlikon, Switzerland. Since 2003, he has led the Chair of Integrated Systems at TUM. His research interests include application-specific multiprocessor architectures, IP network processing, network-on-chip, and self-adaptive fault-tolerant computing.

Prof. Herkersdorf is a Member of the German Research Foundation (DFG) Review Board and serves as editor for Springer and De Gruyter journals for design automation and information technology.