Programmable Logic Controllers in the Context of Industry 4.0

Martin A. Sehr,[1] Marten Lohstroh,[2] Matthew Weber,[2] Ines Ugalde,[1] Martin Witte,[3] Joerg Neidig,[3] Stephan Hoeme,[3] Mehrdad Niknami,[2] Edward A. Lee[2]

***Abstract*—Programmable Logic Controllers (PLCs) are an established platform, widely used throughout industrial automation but poorly understood among researchers. This paper gives an overview of the state of the practice, explaining why this settled technology persists throughout industry and presenting a critical analysis of the strengths and weaknesses of the dominant programming styles for today's PLC-based automation systems. We describe the software execution patterns that are standardized loosely in IEC 61131-3. We identify opportunities for improvements that would enable increasingly complex industrial automation applications while strengthening safety and reliability. Specifically, we propose deterministic, distributed programming models that embrace explicit timing, event-triggered computation, and improved security.**

I. Introduction

WHILE Industry 4.0, digitalization, and the Internet of Things all promise increased use of general-purpose software and networks in industrial applications, there are significant risks. In such applications, safety, reliability, security, and efficiency are even more important than in many information technology and home automation applications. Programmable Logic Controllers (PLCs) provide an ecosystem of relatively simple software logic, robust and ruggedized hardware, networks with controllable real-time behaviors, and extensive availability of interoperable components such as sensors and actuators. As such, PLCs are an established *platform* for factory automation and industrial process design governed by the IEC 61131 standard [1]. The platform includes programming style (in part 3 of the standard), networking style (in part 5), and physical interconnects (in part 2), each enabling composition of components in complex automation systems with predictable behavior. No comparably robust and reliable ecosystem has yet emerged using general-purpose operating systems and networks with embedded software.

Today's PLC ecosystem, however, is suffering growing pains as the complexity of automation systems increases, integration with Internet and wireless services becomes essential, and integration of learning, computer vision, and speech recognition are demanded by end-users. While there have been many innovative changes to PLCs over the past decades, most of the development has been focused on integration of improved hardware components rather than structural changes to the programming model. For example, where early PLCs were able to execute tens of instructions per second, modern PLCs can perform bit operations in nanoseconds. However, the programming model motivated by IEC 61131-3 has not changed substantially across PLC realizations by different vendors. We argue in this article that PLCs as a platform have the potential to grow into the novel capabilities demanded in modern manufacturing problems without compromising their existing advantages. To see how to do that, we discuss essential features of the current platform, identify weaknesses that form barriers, and propose a list of directions for possible improvements. We believe that the suggested adaptations will maintain the benefits of today's PLC designs, particularly safety and reliability, while enabling continued widespread application amid growing requirements.

Whereas simple systems can be designed, prototyped, and tested in their intended deployment context, prototype-and-test design iterations are problematic in industrial automation. Testing low-confidence designs is not an option for complex compositions of components [2], [3]. As a consequence, evolving and augmenting existing designs will require more reliance on formal properties of the platform as well as virtual prototyping, where simulation and verification replace prototype-and-test. It will be important, therefore, when evolving the PLC platform, to not just increase *flexibility* and *generality* of programming possibilities, but also to enforce *constraints* ensuring predictable, analyzable, and reliable behavior.

For example, with some modern PLC models, it is possible to integrate custom C code into otherwise conventional automation code, but with critical constraints. A C procedure that is called synchronously from within a periodic task, for example, will typically have a highly restricted execution environment, limiting access to memory and APIs such as the networking stack. These restrictions help prevent problems in the C code (such as blocking on a network access) from disrupting timing-critical elements of the program. To regain flexibility, these same environments provide mechanisms for asynchronous invocation of arbitrary C code that is much less restricted; asynchronous invocation ensures that invoked C modules cannot disrupt the timing of critical tasks. In more general-purpose programming environments, such isolation is achievable, but it requires considerable expertise and discipline from the programmer. One of the strengths of the PLC platform is that it *enforces* constraints rather than depending

[1]Sehr and Ugalde are with Corporate Technology, Siemens Corporation, Berkeley, CA 94704, USA.

[2]Lohstroh, Weber, Niknami and Lee are with the EECS Department, UC Berkeley, Berkeley, CA 94720, USA. Their work was supported in part by the US National Science Foundation (NSF), award #CNS-1836601 (Reconciling Safety with the Internet)

[3]Witte, Neidig and Hoeme are with Digital Industries, Siemens AG, Nuremberg, 90475, Germany.

Manuscript received FIXME, 2019; revised FIXME, 2020.

on the tenacity of programmers to follow best practices.

In this article, we examine the current state of the practice in PLC-based industrial automation systems, focusing on the essential properties of PLCs that make them robust and reliable. Ultimately, we identify strengths and weaknesses of today's approaches and suggest paths for improvement towards future automation platforms. Our intention with this paper is not to address in detail the various possible solutions to these points, but rather to raise awareness of issues to be addressed in development of the next generation of PLCs to enable continued widespread use throughout industrial automation.

II. PLC Development

A. Historical Perspectives

PLCs were originally designed to substitute electromechanical devices like relays, which implemented simple control logic in cabinets. However, PLCs have evolved to become standardized computers, albeit with some special properties as compared to other control system architectures:

- PLC hardware, unlike many embedded systems, is not designed for a specific installation space. Instead, a PLC is a modular, reusable, and often ruggedized component intended for installation in cabinets.
- The hardware provides a wide range of largely standardized connection options (see part 5 of [1]), powered by a rich programming and configuration system.
- Instead of a general-purpose operating system, PLCs provide a cyclic and prioritized execution model, including cycle time monitoring, adapted to automation.
- PLCs provide a stable runtime environment with basic functionality that prevents programming errors from compromising the integrity of an industrial control system.
- One need not be an expert software or control engineer to program a PLC in a dependable way, increasing the accessibility of the programming model.

The development of these characteristics was largely complete in the late 1980s. Since then, runtime system and overall execution semantics of the PLC have not changed substantially:

1) The main workload of a PLC is done in scan cycles, with processing control tasks organized in *functions*.
2) Functions operate on an internal region of memory called the *I/O image table*, where inputs and outputs are updated manually or automatically, but I/O occurs only at specified time points such as at the beginning and end of a cycle.

Nevertheless, the requirements for PLC programming have gradually evolved over time. In particular:

1) The number of control tasks per PLC has increased dramatically, increasing the risk of undesirable interactions.
2) The number of applications with differing requirements has grown. Today, PLCs are used in processing plants, production machines, assembly lines, and even ships.
3) The types of control applications to be handled have extended over time from simple logic control with sampling times in the order of seconds to milliseconds to complex control schemas, such as in high-frequency motion control for synchronized drives with sampling times down to the order of microseconds, predictive control, or control based on analysis of complex data.
4) PLCs are part of a real-time network on the factory floor, connecting basic sensors and actuators, distributed intelligent peripheral devices (see, e.g., a recent proposal in [4]), and other industrial control systems such as protection switches, motion control systems, Supervisory Control and Data Acquisition (SCADA) systems, and edge devices.

The PLC community has responded to changing requirements with new programming languages, specialized prioritized scan cycles, and dedicated I/O hardware hiding networking from PLC programmers. Also, processing speeds and memory have increased substantially over the years.

B. Embedded Devices

PLCs and other industrial control devices are often viewed as embedded devices, with the idea that these devices, unlike general purpose computers, are dedicated to performing a specific task. An embedded device is simply a microprocessor that is either programmed directly, without an operating system, or indirectly using any of a range of operating systems, including variants of Linux [5], [6]. Such devices can reside on customized printed circuit boards that are adapted to very particular physical settings. For relatively large volume applications, such as controllers in cars, large, complex engineering methodologies have evolved employing model-based design, code generation, and hardware-in-the-loop testing.

In contrast, PLCs are typically sold off-the-shelf, with variants adapted to a wide range of environmental conditions and spanning various protection classes and safety integrity levels. They enjoy flexible but standardized I/O-connectivity. The operating system of a PLC is customized to real-time control, making it easier to make reliable and robust controllers. Moreover, their programming environments are designed for use by specialists in automation, not software. It is cost effective to use a PLC for individual automation problems, where only one instance of the design is expected to ever be deployed. It is also increasingly common for PLCs to coexist and interact with embedded controllers.

C. CPS and IoT

Trends such as Industry 4.0 predict the use of decentralized control and increased intelligence [7], [8]. Today, PLCs are centralized controllers communicating with sensors and actuators as part of the traditional *automation pyramid* (see Fig. 1). All connected devices are viewed as *I/O* type devices, without knowledge of their inherent complexities and behaviors aside from delivered and consumed values.

In decentralized control, communication may no longer be periodic but event-based, and ensuring deterministic results can impose additional requirements such as timestamping on the event-based system. Furthermore, if intelligent systems exchange information, it is imperative to understand at least parts of their mutual behaviors for exhibiting correct reactions. This can be captured formally by contracts [9]. However,

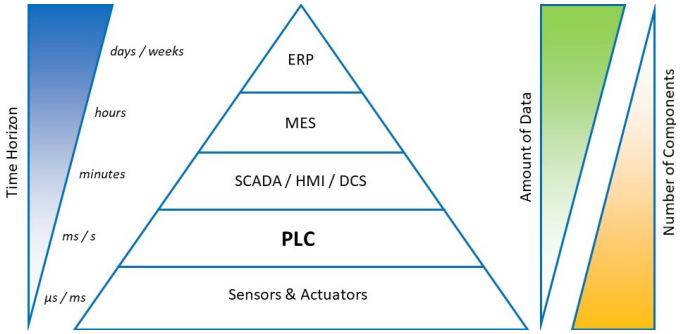

Fig. 1. PLCs form the control layer of the traditional *Automation Pyramid*, operating between field devices, such as sensors and actuators, and supervisory layer systems, such as SCADA, Distributed Control Systems (DCS), or Human-Machine Interfaces (HMI). At the top layers, this data acquired within lower layers is processed by Manufacturing Executions Systems (MES) and Enterprise Resource Planning (ERP).

assumptions on the behavior of peripheral devices and physical systems in current automation code are typically implicit.

The essential role of PLCs, in contrast to higher levels of the automation pyramid, is to interact with sensors and actuators. PLCs, therefore, lie at the boundary between the cyber and the physical in CPS and between the Internet and the Things in IoT. Their design, therefore, is not just a matter of computation and algorithms, but also a matter of physical dynamics. Timing becomes much more than simply a measure of performance; it becomes central to correct operation.

D. Automation of Automation

Automation engineers often lack expertise in control theory, so automation functions may be simple from a control-theoretic standpoint. But automation of a plant is far from trivial. The number of interacting components ranges from thousands in an automotive production line to tens of thousands in the process industry. Moreover, physical side-effects, interlocking, error handling, start-up, shut-down, special commissioning testing, clean-up, functional safety, and manufacturing execution system communication are all aspects to be considered in addition to control of the production process.

Integrated development environments and simulators facilitate the design of complex hierarchical automation systems, but the industry is ripe for the development of more sophisticated computer-aided design techniques. The means for structuring PLC code have not kept up with the growing complexity. Although the programming model has been extended with object orientation in the latest version of IEC 61131-3 [1], the programming model lacks sufficient means for code and data encapsulation, hardware abstraction, layered architectures, and separated process spaces. Moreover, communication mechanisms between code fragments remain primitive, relying largely on shared memory locations and parameter lists.

Because PLCs operate at the boundary between the cyber and the physical, their programming model must necessarily draw from both sides. This is done today with an emphasis on periodic computation with specified timing, but other aspects of the computational model, such as communication between software components, today fail to embrace the physical side. This can be done, for example, with explicit timestamping of messages, as we describe below. Languages and tools that more directly embrace the physical aspects of the design will lie cognitively closer to way of thinking of automation engineers, rather than software engineers, and therefore may be better suited for use on a factory floor.

III. Programmable Logic Controllers

The programming model for PLCs, loosely defined by the standard IEC 61131-3 [1], lies at the heart of their character, so it is worth reviewing here. We highlight its fundamental strengths and weaknessess in order to set the stage for a discussion (see Sec. IV-D) in which we identify opportunities for innovation in the design space of PLCs. We do not address in detail IEC 61499, an event-driven extension of IEC 61131-3, predominantly because it has not achieved significant usage in industry. For perspectives on the differences between IEC 61131-3 and IEC 61499 and, software engineering in industrial automation in general, see [10]–[12].

Compared to general embedded control systems, PLCs provide a more structured and constrained framework for design, with specific support for vetted, commonly-used design patterns [13]–[15]. They can be programmed in a number of languages at different abstraction levels, such as:

- Structured Text (imperative, based on PASCAL);
- Instruction Lists (akin to assembly language);
- Ladder Diagrams (based on ladder logic, a notation used for hardwired relay circuits);
- Function Block Diagram (a graphical language); and
- Sequential Function Charts (graphical, akin to Petri Nets).

We next define three major components of a PLC-based design: *computational components*, consisting of software to be executed; *data*, sections of memory with particular roles; and *devices*, providing data to the computations or use data provided by the computations.

A. Computational Components

We next review the core elements of the programming model defined by IEC 61131-3.

1) Tasks: Tasks are blocks of computation that are executed in response to CPU events and often invoked on a periodic basis via timer interrupts. There is always a *main task* that executes in an infinite loop and may be preempted by other, higher priority tasks. Every task has a notion of a single, finite execution called *cycle*, and cycles of a task may be executed repeatedly, often with an unbounded number of executions.

The main task may also be assigned a *minimum cycle time* T, in which case, if it finishes a cycle before time T has elapsed since the cycle began, then the next cycle is delayed until T has elapsed and the resulting CPU idle time may be used for purposes such as communication, which generally is handled outside the user program. For illustration, Fig. 2 depicts the main cycle with minimum cycle time being preempted by a higher priority task. If the main task is not assigned a minimum cycle time, then it executes *as fast as possible* (AFAP), in that each cycle begins as soon as the previous cycle has ended. Note that only the lowest-priority

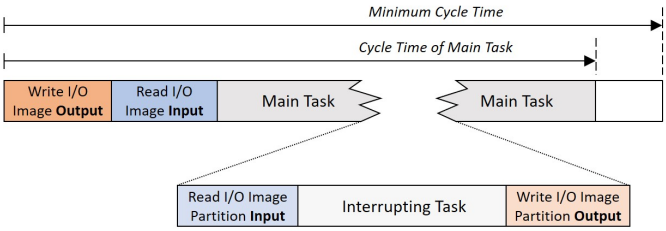

Fig. 2. Main task with minimum cycle time interrupted by a higher priority task; I/O image partition associated with interrupting task is updated accordingly; Main I/O image update prior to execution of main task.

tasks on any single PLC can use this AFAP style because such a task will block execution of any other tasks of lower priority.

Once a cycle in a task is started, it runs to completion, possibly with temporary preemption by a higher priority task. All tasks have distinct priorities, and no task executes unless all other higher-priority tasks on the same PLC are stalled between cycles. Note that this property makes it more difficult to leverage the parallelism of multiple cores in a multicore CPU. Any simple strategy of simply executing tasks simultaneously on multiple cores will violate this property and therefore change program behavior in unpredictable ways.

Consider the sketch of a schedule with four tasks shown in Fig. 3. The horizontal axis is time in some arbitrary units. The main task (at the bottom) has the lowest priority. It uses the processor only when other tasks are idle. In this sketch, the highest priority task (at the top in green) has a period of one time unit and a worst-case execution time (WCET) of 0.5. The sketch assumes that the task actually uses the full WCET, although typically it will use less time than that, often a lot less. The highest priority task always preempts other tasks and hence executes with a regular schedule. For all other tasks, the schedule is irregular and dependent on the actual execution times (not the WCET) of all higher priority tasks.

In the sketch, the second highest priority task (in red) has a period of four time units and a WCET of one time unit. Again, the sketch assumes its execution occupies the entire WCET, in which case it will be preempted once per cycle. The next highest priority task (in blue) is assumed to have a period of eight time units and a WCET of 1.5. In the worst case, this leaves 0.5 time units for the main task in each major cycle of eight time units.

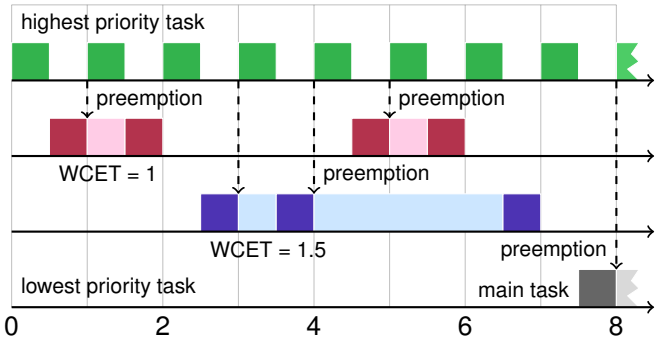

Fig. 3. A nested schedule of periodic tasks.

With this sketch, we can understand a number of serious challenges posed by the PLC model of computation. First, the *de facto* standard practice of using polling to read sensor data requires the period of the task doing the polling to be smaller than the required response time. For example, if we have a sporadic sensor that detects a rare condition and requires a response within, say, 100 microseconds, then this task cannot have a period longer than 100 microseconds. This is true even if the sensor only very rarely produces events. This might seem harmless because, presumably, if the sensor has not detected an event, then the task will not have much to do and therefore will not occupy much execution time. But the real cost is that it will preempt whatever task is currently running.

Preemption has several costs. First, it comes with considerable overhead (saving and restoring processor state). Second, it generates traffic on shared resources such as busses and networks. Third, on almost all modern microprocessors, it invalidates WCET analysis. WCET analysis on modern microprocessors usually requires that interrupts be disabled [16], implying no preemption. Although our sketch implies that preemption occurs at predictable points in the execution, this is not actually the case because the sketch assumes that everything is actually running as slow as possible, occupying its full WCET. This will rarely be the case. As a consequence, preemption occurs at an arbitrary point during execution of a task. This affects caches, branch predictors, and other state-dependent hardware in unpredictable ways. As a consequence, we are forced to make pessimistic assumptions (e.g., that every memory access is a cache miss), resulting in extremely conservative WCET estimates. To ensure safety in such a situation, we have to over provision the system by a considerable margin.

Most or all of these costs can be mitigated with a model of computation that more directly embraces event-triggered rather than just periodic computation, that is compatible with multicore execution, and that can target timing-predictable processors such as PRET machines [17], [18].

On modern PLCs, besides the main, minimum-priority task, both cyclic and non-cyclic tasks within the user program can be defined that will be invoked according to a number of criteria, such as:

1) at specified times of day;
2) after time delays from trigger functions have passed;
3) at specified frequencies and phases from start-up;
4) triggered by hardware interrupts, e.g. from I/O modules;
5) triggered by network devices; and
6) isochronous interrupts, triggered by network events.

A number of other conditions may trigger task activation, including error or fault conditions, system reset, and software events on other processors (called multiprocessor interrupts). Moreover, *alarms* can occur as a result of unforeseen or erroneous conditions, such as communication failures, program errors, or timing constraint violations. By default, an alarm causes the PLC to halt execution of its cyclic control tasks, and, as with traditional languages such as C++, Python, or Java, code can be provided to catch and handle the alarm. The caveat of having these enriching features is that, by affecting the timing of tasks, they tend to undermine the main advantage of a periodic model of computation: analyzability.

There are a number of significant ambiguities about the timing of task execution that do not appear to be well addressed in the IEC standard nor well-defined in software documentation of commercial products. For example, if external events occur at the same time, the order in which they are received from the network is not recorded, making the response nondeterministic. Moreover, in general, sensor values are not timestamped upon being measured, sent, or received, making it difficult to know the staleness of observed quantities. Such ambiguities need to be addressed in order to be able to create reliable and verifiable PLC software.

2) Functions: As with traditional programming languages, the building blocks of code are *functions* of various forms, divided into ones with or without variables retained between consecutive executions. A cycle of a task is given by functions that may call others. Between cycles, the inputs and outputs of functions are stored in a section of memory called the *I/O image table*, as we discuss in more detail in Sec. III-B. Ideally, a function executes atomically in that its inputs do not change during its execution, its outputs are not visible until its computation is complete, and it has no interaction with other functions or I/O devices except through its inputs and outputs.

In current PLC programming environments, however, it is possible to circumvent this idealized pattern because process memory and data may be read and written by more than one function. Moreover, functions may preempt one another and execute concurrently—although typically not in parallel—and undisciplined sharing of memory can lead to nondeterminism and unexpected behaviors. While this could be mitigated by providing warnings through code analysis tools [19], undisciplined use of shared memory is unfortunately common practice. As the complexity of applications increases, it becomes beneficial to enforce a disciplined use of memory to guarantee the idealized model of atomic, deterministic execution. As long as the underlying language is deterministic and the above constraints for idealized execution are met, then each cycle will be deterministic. Given a set of input values, it defines exactly a set of output values and a new state for its memory.

B. I/O Image Table and Program Data

By default, interaction between PLCs and other devices is through the memory system rather than by directly connecting external devices to the PLC microprocessor. Sensors write into the I/O image table, the PLC computes and writes results to the I/O image table, and when the PLC is done, the outputs are transferred to actuators. In more recent PLC models, I/O image tables may also be partitioned.

In the simplest configuration, there is a single task working on an I/O image table, and, prior to each cycle of this main task, the image is updated with sensor data. Only after the cycle of the main task has completed and before the next cycle begins are the results of the PLC computation transferred to actuators. This strategy ensures that input values used by the PLC computation are stable during the entire cycle and that outputs produced by the PLC will be transferred all at once to actuators once computation is finished.

In more elaborate configurations, the I/O image table may be divided into subimages associated with distinct tasks. Each subimage is updated with input data prior to execution of that task, and outputs are transferred upon task completion. If the relative alignment in time of cycles in different tasks is not well defined, then neither are order and timing of the transfer of commands to actuators, indicating a possible source of nondeterminism. Additionally, there are no constraints preventing a task from reading or writing to the process subimage of a different task in current frameworks. This means that for that task, the I/O image table it is working with may *not* be stable during the execution of a cycle, and that the I/O image table data could change whenever the task is preempted, which could lead to unexpected results.

In many PLC models, functions may also use other portions of system memory, called, for example, *data blocks* in Siemens PLCs [20]. Ideally, these data blocks store variables for use only within a single functions. In practice, however, they are also used for communication between functions. If the order of execution of functions in distinct tasks is not well defined, this may provide another source of inadvertent nondeterminism.

C. Network Communication

From a PLC programmer's point of view, the network is invisible, typically abstracted away by peripheral devices. While this helps focusing on the programming of the individual components, one aspect of communication whose abstraction is generally *leaky* is its timing behavior. Communication occurs either regularly or irregularly, and can be categorized by timing patterns as follows:

- **Periodic (synchronized)**: Components communicate periodically based on a common global reference clock, eliminating synchronization overhead during operation.
- **Periodic (drifting)**: Components communicate periodically, but in accordance to their own local clocks which are initially and/or periodically synchronized to a global reference clock. In this case, local clocks may naturally drift from the reference clock, requiring explicit clock-synchronization logic when interfacing components.
- **Quasi-periodic with time-varying periods**: Communication occurs periodically with respect to a reference clock, but the periods are allowed to vary in response to control signals or other conditions of the system.
- **Sporadic**: Communication is irregular, triggered by occurrence of external events, for instance ones captured by light barriers or temperature sensors.

Functions typically see network messages as inputs through the I/O image table, but reasoning about collective behavior of functions scattered on a network and communicating using these patterns can be difficult. Even the simplest pattern, synchronized periodic, may not be synchronized with the execution of functions. For example, functions within the lowest-priority task using the AFAP policy exhibit unpredictable timing relative to periodic network communication. This may result in messages being missed or processed more than once.

Various components and interconnections of a control system may have their own clocks interfacing with each others' clock domains. Clocks that are synchronized may be synchronized to periods that are multiples of each other, and phase

shifts can be introduced to compensate for message delays, for example in PROFINET Isochronous Real-Time (IRT) mode. The normal synchronization of PLCs is periodic (drifting), but it is possible to synchronize the network completely in IRT mode, so that all communication loops are running in sync. But if IRT mode communication is combined with devices that use their own clocks to define periodic actions, as is common, then the resulting timing behaviors become complex, chaotic, and difficult to analyze.

A question that will become more important as systems switch to Time-Sensitive Networks (TSNs) is the relationship between the timing of isochronous actions, triggered by the network, and the timing of periodic tasks, triggered by timer interrupts based on a local clock. Even in the absence of TSN, it is already common to set up a master clock on a local-area network so that periodic events on multiple PLCs are at least frequency-locked, if not phase-locked.[1] However, delays along network communication channels can create jitter that is difficult to predict. This is problematic because the time delay between the arrival time of a packet and the time at which it is loaded into the I/O image table is not tracked, and these delays can be relatively long compared to the cycle period.

IV. Opportunities

Industrial automation is an understandably conservative business; disruptions to production lines can be costly and significant safety risks have to be managed. At the same time, market, cost, and competitive pressures demand innovation. Complexity and demand for customizability of products within product lines keep increasing in a competitive market. Machinery on the factory floor increasingly needs to be connected to networks in order to leverage improvements in condition-based maintenance, energy optimization, lean supply-chain management, and coordination across departments. To respond to these pressures, facilities must evolve, but they must do so in a minimally disruptive way: new equipment must be deployed with minimal disruption to existing production, and new configurations and interoperability of legacy and new equipment must be tested prior to deployment.

These requirements demand important changes to PLCs: they must operate safely in open networking environments; they must be testable in virtual prototypes; and the behavior of their software must be more independent of the hardware so that new hardware can be deployed without disrupting existing functions. We believe that these requirements call for some crucial changes to the computational models that today form the core of PLC design.

A. Timing Requirements

Future PLC designs must rely less on priority-based cyclic execution models whose timing depends on unrelated tasks running on the PLC or elsewhere in the network. Instead, PLC designs should specify timing behaviors, such as deadlines, and hardware and operating system infrastructure should ensure that the behavior is as specified. This implies less reliance on priorities because, given only priorities, the actual behavior of one component depends on other, unrelated components. Instead of priorities, software components should specify timing requirements and the compilers and operating systems should ensure that these timing requirements are met.

A clean model of time would make PLC-based designs more testable and more faithful to virtual prototypes. Ideally, timing should be a logical property of programs as well as a physical property of their implementation. The notion of logical synchrony, for example, can be used even on physically asynchronous systems [21]. Two events are logically synchronous if no external observer can see that one event has occurred and the other has not. Implementing logical synchrony does not require that events actually occur simultaneously. Instead, such synchrony can be realized by controlling what observers can see, enabling a path to ensure that periodic actions are coordinated in a predictable, repeatable and testable way.

We further argue that making a commitment to determinism can improve testability and safety of systems. For example, it is common among control engineers to always want to use the most recent measurements from sensors. In complex systems, however, a component may combine recent data from one sensor with stale data from another, thereby building an inconsistent view of the physical system state. Timestamping data can help, particularly if timestamps are interpreted as a logical property of programs. An execution environment that ensures that every software component sees messages only in timestamp order can go a long way towards making behaviors both more understandable and predictable.

Physical time is easy to define for a single PLC implemented on a microprocessor with a single real-time clock; for such a system, physical time is simply the time revealed by that real-time clock. For a multi-PLC system, however, physical time is harder to define precisely, so we must not rely on it to give semantics to the system. To achieve deterministic computation on such a distributed system, the implementation will have to coordinate physical time measurements across the system, using, for example, clock synchronization protocols [22]. No clock synchronization mechanism is perfect, but if there is a known bound on the discrepancy between clocks, then it is possible to enforce a consistent logical notion of time across a distributed system [23].

Another requirement is a common logical time origin, by which we mean that all tasks are launched logically simultaneously at the beginning of the execution of the application. Hence, two periodic tasks A and B that have the same period P have logically simultaneous cycles, although the actual order of execution of their cycles will depend on their priorities (if they are executing on the same CPU) or on scheduling (if they are executing parallel). To maintain logical simultaneity, all that is required is that any observer that has seen n executions of A has also seen n executions of B.

On a single CPU, a notion of logical synchrony of periodic tasks is relatively easy to maintain. We can ensure that during execution, the number of cycles completed by A does not differ by more than one from the number of cycles completed

[1] "Frequency locked" means that if two periodic tasks A and B with the same period execute on two different PLCs, then the difference between the number cycles that A has executed and the number of cycles that B has executed remains bounded at all times.

by B. Moreover, all tasks with lower priorities than both A and B or higher priorities than both A and B can never observe any difference in the number of cycles of A and B that have been executed. In this sense, A and B are logically simultaneous. A task with priority between those of A and B will always observe a difference of exactly one between the number of invocations of A and B. If A and B have the same priority, then the order of their execution can be determined by data precedences, if one uses data computed by the other, or can be arbitrary, if there is no interaction between them.

Once there is a notion of logical simultaneity, it becomes possible even for the absence of an event to convey information. At a logical time instant, either two events are present simultaneously, one is present and the other is absent, or both are absent. Such semantics is realized in synchronous languages such as SCADE [24], which is used in safety-critical avionics software. Such software has a rigorous meaning that can be modeled and analyzed mathematically. Consider a situation where A sends data to B in each cycle of periodic execution, and execution of B depends on execution of A having been completed first. Such dependencies can be handled in interconnected components that execute logically simultaneously if the data exchanged is semantically the fixed-point of a monotonic function in a generalized metric space [25]. Moreover, the implementation of such semantic models can be extremely efficient [26], at least on a single computer.

Maintaining such logical synchrony in a multicore or distributed system implementation is more challenging, but realizable by leveraging synchronized clocks [23]. Global logical synchrony may be, on the other hand, excessively restrictive for some applications. For such applications, we could introduce logical clock domains, as done in [27]. Logical clock domains can provide islands of synchrony where interactions across the islands are asynchronous.

B. Deterministic Execution & Parallelism

The designs should also be more deterministic, by which we mean that the response to a given set of input conditions should be defined by the software and be unique. Specifically, a response should not depend on how clocks are drifting with respect to one another nor on detailed execution times of software on the PLC. Determinism improves testability: defining a single correct response to a set of input conditions means that those input conditions can be used to test the system and help enable virtual prototyping. For the same reasons, these changes also make programs more independent of deployment hardware; if the hardware on which the software executes is updated, system behavior will be unaltered provided the new hardware can deliver correctly the specified timing.

Most PLCs today are realized by software on commercial off-the-shelf microprocessors. Today, most microprocessors have multiple cores, something that traditional PLC programming models do not easily accommodate or benefit from. Our suggestions of extending or changing the programming model to address timing specifications and determinism would also allow ensuring any revised programming model is able to effectively and safely exploit multiple cores, requiring better mechanisms for software components to interact with one another. Using shared variables in memory, for example, can work well when execution of functions is atomic and mutually exclusive, but if functions are executed in parallel on multiple cores, the behavior may be affected by uncontrolled low-level timing effects. Message-passing communication mechanisms, among others, can mitigate this risk, preserving determinism while allowing for parallel execution.

C. Event Handling

The current cyclic execution semantics and common I/O image table of a PLC can, because of their simplicity, create issues for event-based control problems. For such problems, many functions start with a *has-something-changed* code fragment before processing their logic, conceptually recreating an event-based system in an ad-hoc manner via polling. Excessive polling, however, can overload the network, which prevents the ability to use the network for other purposes. For rare events, this becomes particularly inefficient.

Any event that is handled by a cyclic task with period T has a worst-case response time of at least T. If an event requires a quick response, then T must be small, meaning that the frequency of the task must be high. This can lead to increased processing load and network traffic. Moreover, if any two events that are mapped to the same cyclic tasks with period T, then the order which these events occur becomes ambiguous if they occur within time period T. For events where order is important, this could again lead to an excessively small period T. When information is exchanged across non-synchronized networks, cyclic processing of events can increase delays due to cycle misalignments. The worst-case latencies add.

Events, especially rare ones, could be handled better by an event-based rather than a polling-based programming model (see also [28]). Indeed, an event-based style was attempted in IEC 61499 (see for example [29]), but the programming model proposed in this standard proved to be nondeterministic [30] and too complicated to gain wide adoption in industry. We should not give up, however, because of an unsuccessful standard. Event-based scheduling should become intrinsically supported by peripherals and should be performed at a granular level in future PLC software.

We believe that alternative event-based programming models should be explored, particularly those that can be combined with traditional periodic task execution. The so-called "reactor" model of computation, introduced in [31], uses timestamped events and deadlines to combine periodic and sporadic tasks in such a way that latencies can be controlled without resorting to polling. This model of computation, if combined with processor architectures with controllable timing (so-called PRET machines [17], [18]), can even deliver deterministic latencies on combinations of sporadic tasks without increasing worst-case latencies [32].

The reactor model of computation has recently been realized as a coordination language called Lingua Franca [33]. This realization looks particularly promising for the PLC context because it reportedly delivers extremely high performance (up to 2.5 million reactions per second per core on a laptop

computer), it is able to exploit multiple cores without sacrificing determinism, it is able to run on platforms with little or no operating system infrastructure, and the programming functionality can be written in C, perhaps the most portable and low-overhead language in use today [33].

According to [26], [34], the reactor model of computation is also well suited to distributed execution on network-connected computers using a technique called PTides [23]. PTides is based on the same discrete-event (DE) model of computation as Lingua Franca, where communication between components occurs via timestamped messages. DE models have been shown to be deterministic, even in distributed settings. Moreover, they generalize synchronous-reactive models [35], [36], which have been used in safety-critical avionics system designs [24], [37] and excel at periodic task execution. Moreover, the PTides technique has been shown to scale even to globally distributed systems by Google, which uses the technique in a distributed database systems called Spanner [38].

D. Network Access & Communication

Assuming time-sensitive networking (TSN) gains traction in industry [39], additional opportunities present themselves: high precision synchronized clocks could enable better coordination of code running on separate PLCs; time-slotted, reservation-based network traffic could improve determinism; time-sensitive traffic shaping could improve safety. This would expand on existing industrial Ethernet protocols such as PROFINET, which already provide a level of deterministic message delivery.

If the programming model of PLCs is to change, a number of other improvements should be considered. For example, sandboxing, private memories, and temporal isolation [40] could make programs more composable and even allow execution of unverified code. Another opportunity is to introduce state-of-the-art authentication, authorization, and encryption, a prerequisite to opening networks to outside traffic [41]. To support situations where shared data is required, infrastructure for private memory and local communication, such as via pipes or message-passing, could be provided to make sharing data in memory safer than shared variables. In addition, if messages and sensor data are timestamped as proposed above, then runtime systems can regulate the order of message delivery and prevent stale data from being combined with fresh data [23].

These innovations would allow a change in vision for the PLC from a highly specialized controller on a closed network to a more general platform for specialized control and service composition on an open network. Such a PLC could combine local control with a modern microservice architecture in which loosely coupled cloud services are integrated with the primary control application. For example, a PLC could be connected to a cloud-based speech recognition service to translate spoken commands into changes in high level control modes.

A semantic notion of time and more deterministic execution model can enable the integration of microservices without sacrificing the reliable control of local devices that PLCs are traditionally intended for. While the level of reliability of a cloud-based microservice is not expected to match that of an industrial controller with real-time requirements, careful integration can ameliorate the impact of unexpected behaviors. For instance, in the scenario where a PLC is the client of a cloud-based microservice, it is likely problematic if the microservice were to silently fail. By enforcing deadlines and catching service failures and network timeouts on the client side, these situations can be handled in a predictable manner.

On the networking side, timing also provides regulation of the open aspects of the network. TSN specifically enables such mixed criticality use of the network via open and closed time slices. Similarly, deterministic execution guarantees that microservice requests (but not necessarily responses) are predictably executed. An example of such a component architecture for deterministic microservice composition is found in [42], which primarily targets the Internet of Things. PLCs could benefit from a similar architecture.

E. Virtual Prototyping

A virtual prototype is a software model to be used for simulation or analysis. Virtual prototyping has proven very effective in VLSI design, where physical prototypes are costly. In industrial automation, however, effective virtual prototyping has proven elusive, although recent efforts have made progress on this front. SDCworks [43], for instance, provides a framework for formalizing and verifying controllers for smart manufacturing systems. Another model-based approach is discussed in [44]. A key challenge is that while the behavior of physical machinery is timing dependent, the timing behavior of software is difficult to control. Hence, a simulation model may need to include excessive low-level detail about the implementation, rendering analysis intractable and simulation slow or impossible.

In [45], Lee and Sirjani distinguish what they call scientific models, which are intended to reflect the behavior of preexisting systems, from what they call engineering models, which are intended to specify the behavior of a system to be built. They point out that it is important to recognize whether a model is to be used for a scientific or an engineering purpose. An engineering model can serve as a *specification*, a detailing of requirements that a physical realization must satisfy. If the engineering model is built in such a way that its requirements can be met in a cost effective way, then it can be used to validate a design before any physical prototype is constructed. In VLSI design, a VHDL or Verilog program is an engineering model of a chip and almost all verification and validation tasks can be carried out on the model without constructing physical hardware. The challenge we pose here is to make engineering models of industrial automation systems as effective.

A scientific model of a factory automation system, in contrast, needs to model the timing of software execution in great detail if that timing causes significant effects on the physical plant. For example, timing actions running AFAP will depend on every detail of the microprocessor implementing the PLC, its pipeline, memory architecture and I/O system. On the other hand, a minimum-priority task with a minimum cycle time can be effectively modeled at much higher level.

The interactions between the software and the physical plant can even be made deterministic under certain assumptions.

Deterministic models are easier to validate. A simulation of a virtual prototype under specified inputs, if the model is deterministic, reveals the one unique behavior that will be expected under those inputs. For such a deterministic model to be faithful to the physical realization, the following assumptions must be satisfied:

1) First, all periodic tasks must complete execution within one period. Any task that has a *minimum* cycle time (rather than a period) will also have to complete execution in each iteration before that minimum cycle time expires. This makes the task periodic. Variations in execution time will be masked.
2) In order to guarantee (1), we will need to determine the worst-case execution time (WCET) of each task. Despite decades of progress, such analysis can be challenging in practice and may involve unrealistic assumptions [16], such as an assumption that interrupts are disabled. But disabling interrupts of lower priority tasks is clearly not a good idea (see Figs. 2 and 3). An alternative approach may be to implement PLCs on top of PRET machines, which have been shown to be able to support interrupts without invalidating WCET analysis [17], [18].
3) More than just verifying individual task WCETs, we must verify that all combinations of periodic and sporadic tasks that are allowed by the software remain *feasible*, meaning that deadlines are met. For periodic tasks, the deadline must be no longer than the period. If WCETs are known, proving feasibility is challenging. One possible analytical approach that works at least for some models uses network calculus [46].
4) Deterministic interaction between software and its physical environment requires controlling not just when sensing is performed but also when actuation is performed. There is a tendency in many applications to perform actuation as soon as the software knows what actuation needs to be performed, but then the timing of such actuation is difficult to control. An alternative is to delay actuation until a pre-specified time (which becomes a deadline for the software) or until the end of a cycle for periodic actuation without deadlines. Such delayed actuation is anathema to many automation engineers, who are guided by the mantra that low delay in feedback control loops is always preferable. However, delayed actuation can eliminate timing jitter in actuators, which could reduce wear on physical components and make behavior more repeatable and testable. Moreover, in a safety-critical system, we have to validate the behavior of the system under worst-case timing conditions, so we have to design the system to work with worst-case delays anyway. Finally, the biggest benefit of delayed actuation may be the resulting simplicity in simulation and analysis. Immediate actuation requires detailed modeling of the software timing, an extremely challenging proposition.

The Lingua Franca language described in Sec. IV-C is explicit about periods and deadlines, and hence holds promise as a specification (an engineering model) against which an implementation can be evaluated. Even if Lingua Franca does not prove to be the ideal programming model for industrial automation, it embodies an existence proof for effective virtual prototyping. We strongly believe that there are opportunities for development of more specialized, domain-specific programming models closer to current PLC practice while enabling effective virtual prototyping.

V. The Future of PLCs

The trend of technology evolving towards more sophisticated networks, multicore architectures, and increasingly complex microprocessor architectures, poses formidable challenges to current PLC platforms. The specialized programming model used in PLCs makes it difficult to accommodate and take advantage of these new technologies, and attempts to do so tend to compromise the ability guarantee safety. Verifying and simulating PLC designs that use these modern technologies has become impossibly complicated. To address this tension, we believe a paradigm shift is necessary. The desirable properties of future PLCs that we have outlined include:

- Programs specify timing, not priorities.
- Timing as a logical notion (logical synchrony).
- Deterministic concurrency and multicore execution.
- Event-triggered computation as well as periodic.
- Synchronized clocks.
- Common logical time origin across a distributed system.
- Logical clock domains.
- Message passing to replace shared variables.
- Less reliance on polling.
- Sandboxing.
- Private memories.
- Temporal isolation.
- Authentication and authorization.
- Encrypted communication.
- Mixed criticality networking.
- Faithful virtual prototypes.

VI. Conclusion

PLCs are an old but tenacious technology. They survive in part because general-purpose software technology fails to deliver properties that are essential to industrial automation, such as precise control over timing. The programming model for PLCs, however, needs to evolve to support the increasing complexity and more extensive network integration demanded by Industry 4.0. This evolution cannot compromise on safety requirements, which remain the highest priority.

We believe it is time to reexamine the core PLC programming model with an eye towards achieving determinism, enabling virtual prototyping, taking advantage of multicore architectures, leveraging networking innovations like time-sensitive networks (TSN), and strengthening safety guarantees. We have pointed to some proofs that such programming models exist, and we hope that this article will inspire further innovation in this direction.

REFERENCES

[1] International Electrotechnical Commission, *International Standard IEC 61131: Programmable Controllers*, 4th ed. IEC, 2017.

[2] A. Guignard, J.-M. Faure, and G. Faraut, "Model-based testing of PLC programs with appropriate conformance relations," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 350–359, 2017.

[3] W. Wang, N. Niu, M. Alenazi, and L. Da Xu, "In-place traceability for automated production systems: A survey of PLC and SysML tools," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3155–3162, 2018.

[4] H. Pearce, S. Pinisetty, P. S. Roop, M. M. Kuo, and A. Ukil, "Smart I/O modules for mitigating cyber-physical attacks on industrial control systems," *IEEE Transactions on Industrial Informatics*, 2019.

[5] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011.

[6] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*, 2nd ed. MIT Press, 2017.

[7] L. Da Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[8] L. Li, S. Li, and S. Zhao, "QoS-aware scheduling of services-oriented internet of things," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1497–1505, 2014.

[9] P. Derler, E. A. Lee, M. Törngren, and S. Tripakis, "Cyber-physical system design contracts," in *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, April 2013, pp. 109–118.

[10] K. Thramboulidis, "IEC 61499 vs. 61131: A comparison based on misperceptions," *arXiv preprint arXiv:1303.4761*, 2013.

[11] A. Zoitl and V. Vyatkin, "Different perspectives: Face to face; IEC 61499 function block model: Facts and fallacies – IEC 61499 architecture for distributed automation: The 'glass half full' view," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 7–23, 2009.

[12] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, 2013.

[13] W. Bolton, *Programmable logic controllers*, 6th ed. Newnes, 2015.

[14] R. W. Lewis, *Programming industrial control systems using IEC 1131-3*. IET, 1998.

[15] J. W. Webb and R. A. Reis, *Programmable logic controllers: principles and applications*, 5th ed. Prentice Hall PTR, 2002.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing *et al.*, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[17] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Design Automation Conference (DAC)*, 2007, Conference Proceedings.

[18] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *Real-Time and Embedded Technology and Application Symposium (RTAS)*, 2014, Conference Proceedings.

[19] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger, "Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 1, pp. 37–47, 2016.

[20] H. Berger, *Automating with SIMATIC: Controllers, Software, Programming, Data Communication, Operator Control, and Process Monitoring*, 3rd ed. Publicis Corporate Publishing, 2006.

[21] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. Ölveczky, "PALS: Physically asynchronous logically synchronous systems," Univ. of Illinois at Urbana Champaign (UIUC), Report, 2009. [Online]. Available: http://hdl.handle.net/2142/11897

[22] J. C. Eidson, *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.

[23] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Distributed real-time software for cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 45–59, 2012.

[24] G. Berry, "The effectiveness of synchronous languages for the development of safety-critical systems," Esterel Technologies, Report, 2003, Overview of SCADE/Lustre. [Online]. Available: http://www.esterel-technologies.com

[25] A. Cataldo, E. Lee, X. Liu, E. Matsikoudis, and H. Zheng, "Discrete-event systems: Generalizing metric spaces and fixed point semantics," EECS Department, University of California, Report Technical Report UCB/ERL M05/12, April 8 2005.

[26] M. Lohstroh and E. A. Lee, "Deterministic actors," in *2019 Forum for Specification and Design Languages, FDL 2019, Southampton, United Kingdom, September 2-4, 2019*. IEEE, 2019, pp. 1–8.

[27] C. Jerad and E. A. Lee, "Deterministic timing for the industrial internet of things," in *IEEE Int. Conf. on Industrial Internet (ICII)*. IEEE, 2018, Conference Proceedings.

[28] M. Chmiel and E. Hrynkiewicz, "An idea of event-driven program tasks execution," in *IFAC Proceedings Volumes*, vol. 42, no. 1. Elsevier, 2009, pp. 17–22.

[29] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[30] G. Cengic, O. Ljungkrantz, and K. Akesson, "Formal modeling of function block applications running in IEC 61499 execution runtime," in *11th IEEE International Conference on Emerging Technologies and Factory Automation*, 2006, Conference Proceedings.

[31] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee, "Actors revisited for time-critical systems," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 152.

[32] E. A. Lee, J. Reineke, and M. Zimmer, "Abstract PRET machines," in *IEEE Real-Time Systems Symposium (RTSS)*, December 5 2017, Conference Proceedings, invited TCRTS award paper.

[33] M. Lohstroh, M. Schoeberl, M. Jan, E. Wang, and E. A. Lee, "Work-in-progress: Programs with ironclad timing guarantees," in *Proceedings of the International Conference on Embedded Software Companion, New York, NY, USA, October 13-18, 2019*. ACM, 2019, pp. 1–2.

[34] M. Lohstroh, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," in *Cyber Physical Systems. Model-Based Design*, R. Chamberlain, M. Edin Grimheden, and W. Taha, Eds. Cham: Springer International Publishing, February 2020, pp. 59–85.

[35] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *EMSOFT*. ACM, 2007, Conference Proceedings, pp. 114 – 123.

[36] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[37] S. P. Miller, D. D. Cofer, S. Lui, J. Meseguer, and A. Al-Nayeem, "Implementing logical synchrony in integrated modular avionics," in *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, 2009, Conference Proceedings, pp. 1.A.3–1–1.A.3–12, pALS: Physically asynchronous and logically synchronous systems.

[38] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost *et al.*, "Spanner: Google's globally-distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 8, 2013.

[39] M. Wollschlaeger, T. Sauter, and J. Jasperneite, "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0," *IEEE Industrial Electronics Magazine*, vol. 11, no. 1, pp. 17–27, 2017.

[40] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Design Automation Conference (DAC)*. ACM, 2011, Conference Proceedings.

[41] H. Kim and E. A. Lee, "Authentication and authorization for the Internet of Things," *IT Professional*, vol. 19, no. 5, pp. 27–33, 2017.

[42] C. X. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber, "A component architecture for the internet of things," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1527–1542, 2018.

[43] M. Potok, C.-Y. Chen, S. Mitra, and S. Mohan, "SDCworks: A formal framework for software defined control of smart manufacturing systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 88–97.

[44] E. Estevez and M. Marcos, "Model-based validation of industrial control systems," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 302–310, 2011.

[45] E. A. Lee and M. Sirjan, "What good are models?" in *Formal Aspects of Component Software (FACS)*, vol. LNCS 11222. Springer, 2018, Conference Proceedings.

[46] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, ser. Lecture Notes in Computer Science (LNCS), G. Goos, J. H. Hartmanis, and J. van Leeuwen, Eds. Springer-Verlag, 2001, vol. 2050.

Martin A. Sehr Martin A. Sehr received his PhD in Dynamic Systems and Control from UC San Diego in 2017. Since then, he has been working on research in industrial automation and robotics.

Marten Lohstroh Marten Lohstroh received a B.S. degree in computer science and an M.S. degree in grid computing from the University of Amsterdam. He is currently a Ph.D. Candidate at UC Berkeley. His research is focused on deterministic models of concurrent computation.

Mathew Weber Matthew Weber is a postdoctoral researcher at UC Berkeley. He earned a B.S. in Computer Engineering from the University of Virginia and a PhD in Computer Science from UC Berkeley. His research focus is on cyber-physical systems, in particular technologies for modeling real-world context and coordinating the interaction of connected devices.

Ines Ugalde Ines Ugalde is a Research Scientist at Siemens Corporation, Corporate Technology in Berkeley, CA. She has received a B.S. in Telecommunications Engineering from the University of the Basque Country and a M.Sc. in Electrical and Computer Engineering from Rutgers University, where she continues to be affiliated as a PhD Candidate. Her academic work is focused on connected vehicles, wireless signal propagation modeling and communication simulation. At Siemens Corporation, she has focused on robotic vision, manipulation and artificial intelligence for manufacturing processes. Her work has been showcased at several international industrial trade fairs.

Martin Witte Martin Witte is a Senior Principal key expert at Siemens AG, Germany. He has received his M.Sc. in Mathematics at Syracuse University, a Diploma in mathematical economics and a Dr. rer nat. in Mathematics at the University of Ulm. He has been at Siemens AG for almost 30 years, working in the areas of Simulation, Software and System Engineering, and Industrial Automation. He holds multiple patents in this areas. His latest interest is in semantic methods for engineering environments.

Stephan Hoeme Stephan H ome is a research engineer at Siemens AG, Germany. He received a Diploma in Electrical Engineering and a Dr.-Ing. in Automation Engineering at University Magdeburg. He is working in the area of industrial automation, his current focus is on industrial communication systems.

Mehrdad Niknami Mehrdad Niknami is a graduate student researcher at UC Berkeley. His interests intersect various areas of computer science, including programming languages, information security, and traffic engineering. His most recent research has been on improving the programming model of control systems used in industrial automation.

Edward A. Lee Edward A. Lee has been working on embedded software systems for 40 years. After detours through Yale, MIT, and Bell Labs, landed at Berkeley, where he is now Professor of the Graduate School in EECS. His research is focused on cyber-physical systems. He is author of leading textbooks on embedded systems and digital communications, and has recently been writing books on philosophical and social implications of technology.