Proc. of 21st IFAC World Congress, Berlin, Germany, July 12-17, 2020.

Opportunities for Industrial Control

Martin Witte [*] **Martin A. Sehr** [**] **Ines Ugalde** [**]
Joerg Neidig [*] **Mehrdad Niknami** [***] **Stephan Hoeme** [*]
Edward A. Lee [***]

[*] *Siemens AG, Nuremberg, Germany.*
[**] *Siemens Corporation, Berkeley, CA 94704, USA.*
[***] *Department of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA 94720-1770, USA.*

Abstract: Programmable Logic Controllers are an established platform used throughout industrial automation, but rather poorly understood among researchers in the control systems community. This paper gives an overview of the state of the practice in industrial control systems while presenting a critical analysis of the dominant programming styles used in today's automation systems. We describe the patterns standardized loosely in IEC 61131-3 and, where there are ambiguities in the standard, realized in concrete vendor implementations. Ultimately, we suggest directions for further research towards enabling increasingly complex industrial control applications subject to the novel requirements of Industry 4.0 settings without compromising the safety and reliability guaranteed by the current industrial automation stack.

Keywords: Industrial Control Systems, Programmable Logic Controllers, Automation, Industry 4.0, Flexible Manufacturing Systems.

1. 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, International Electrotechnical Commission (2017). The characteristics of this platform include programming style (Part 3 of the standard), networking style (Part 5), and physical interconnects (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, vision and speech recognition are demanded by end-users. We argue in this article that PLCs as a platform have the potential to grow into such capabilities 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.

Where simple systems can be designed, prototyped, and tested in their intended deployment context to iterate designs, such design iterations are problematic in industrial automation, where testing low-confidence designs is not an option, and systems are often complex compositions of many components. As a consequence, growing in complexity and evolving existing designs will require more reliance on formal platform properties 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*, but also to enforce *constraints* ensuring predictable, analyzable, and reliable behavior.

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. We hope to encourage further work on the rich set of topics identified below through the control and automation communities.

2. PROGRAMMABLE LOGIC CONTROLLERS

The programming model for PLCs, which is loosely defined by the standard IEC 61131-3, International Electrotechnical Commission (2017), is the heart of their character, so it is worth reviewing here. We propose here extensions of the standard and also adaptations that, while not modifying the standard, suggest using particular patterns of design among the many possibilities allowed by the standard. [1]

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. 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);
- Sequential Function Charts (graphical).

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.

2.1 Computational Components

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.

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, Figure 1 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 tasks on any single PLC can use this AFAP style because such a task will block execution of any other tasks of lower priority.

Besides the main, minimum-priority task, cyclic and non-cyclic tasks within the user program can be defined that will be invoked according to 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. via I/O;
(5) triggered by network devices;
(6) isochronous interrupts, triggered by network events.

A number of other conditions may trigger task activation, including error or fault conditions, system reset, and

[1] We do not address IEC 61499, an event-driven extension that has not gained significant industry usage. For a broader view on software engineering in industrial automation, see e.g. Vyatkin (2013).

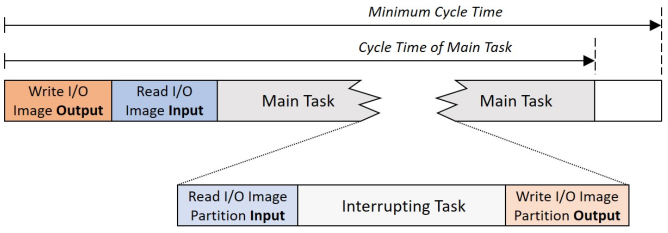

Fig. 1. Main task with minimum cycle time interrupted by a higher priority task; I/O image partition associated with interrupting task is updated accordingly; Main I/O image update prior to execution of main task.

software events on other processors. Moreover, *alarms* can occur as a result of unforeseen or erroneous conditions. 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. Once a cycle in a task is started, it runs to completion, but not necessarily without temporary preemption by another higher priority task. All tasks executing on a PLC 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.

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 and verify that PLC software is behaving correctly and reliably.

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 typically specified by functions, which may in turn 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 below. 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. Unfortunately, such undisciplined use of shared memory is common in practice. As the complexity of PLC applications increases, it may become 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.

2.2 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, as in the example shown in Figure 1.

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. This strategy means 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 e.g. *data blocks* for Siemens PLCs in Berger (2006). 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.

2.3 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 clock, eliminating synchronization overhead at operation.
- **Periodic (drifting)**: Components communicate periodically, but in accordance to local clocks which are initially and/or periodically synchronized to a global 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 system conditions.
- **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. 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. The relationships between these clocks and the resulting timing behaviors, however, can be difficult to understand.

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.[2] However, delays along network communication channels can create significant jitter that is difficult to predict. In particular, for example, 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, which can be problematic as these delays can be long.

3. 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

[2] "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.

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 form the core of PLC design.

3.1 Timing Requirements

Specifically, 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, as discussed by Sha et al. (2009). 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 periodic actions that are coordinated in a predictable, repeatable and testable way.

We further observe 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 as in Eidson (2006). 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, as shown by Eidson et al. (2012).

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.[3]

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 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.

Maintaining such logical synchrony in a multicore or distributed system implementation is more challenging, but realizable leveraging synchronized clocks as in Eidson et al. (2012). 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 by Jerad and Lee (2018). Logical clock domains can provide islands of synchrony where interactions across the islands are asynchronous.

3.2 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

[3] This can be relaxed by fixed-point semantics, Cataldo et al. (2005).

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. Our suggestions of revising 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. 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.

3.3 Event Handling

The current cyclic execution semantics and common I/O image table of a PLC can, by means of their simplicity, also create issues for event-based control problems: PLCs *forget* the event order in a cycle, potentially causing unnecessary network traffic and processing time on the PLC. With intelligent field devices, cyclic processing in non-synchronized networks can create even more delays resulting from cycle shifts peripheral to PLCs, while synchronized cycles can decrease the network throughput. Indeed, many functions start with a *has-something-changed* code fragment before processing their logics, conceptually recreating an event-based system in an ad-hoc manner via polling. Excessive polling, however, can overload the network, preventing the ability to use the network for other purposes. Consider for example a sensor detecting an event that requires reaction by the software within τ seconds, so that a periodic task with period no longer than τ seconds must be deployed and communication between the sensor and the PLC must occur periodically with period no larger than τ. If said event is rare, then polling incurs considerable overhead in network traffic and CPU usage.

Events, especially rare ones, could be handled better by an event-based rather than a polling-based programming model. Indeed, an event-based style was attempted in IEC 61499, but the programming model proposed in this standard proved to be nondeterministic and too complicated to gain wide adoption in industry, as discussed e.g. by Cengic et al. (2006). We believe that alternative event-based programming models should be explored, particularly those that can be combined with traditional periodic task execution. An example is PTIDES, Eidson et al. (2012), which is based on a discrete-event (DE) model of computation, 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 as in Lee and Zheng (2007); Benveniste and Berry (1991), which excel at periodic task execution. Event-based scheduling should become intrinsically supported by peripherals and should be performed at a granular level.

3.4 Network Access & Communication

Assuming time-sensitive networking (TSN) gains traction in industry (see e.g. Wollschlaeger et al. (2017)), additional opportunities present themselves: high precision synchronized clocks could enable better coordination of 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 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 infrastructure, a prerequisite to opening networks to outside traffic (see Kim and Lee (2017)). In addition, messages and sensor data could be timestamped to regulate the order of message delivery as in Eidson et al. (2012). 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.

3.5 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. One 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.

Lee and Sirjan (2018) 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 in a scientific or an engineering way. An engineering model can serve as a *specification*, a detailing of requirements that a physical realization must satisfy. If the engineering model is built so 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. Alternatively, a minimum-priority task with a minimum cycle time can be effectively modeled at much higher level. Its interactions with the physical plant can even be made deterministic under certain assumptions.

First, we must assume that execution time never exceeds the minimum cycle time. This may be guaranteed using execution time analysis, even though such analysis can be challenging in practice and may involve unrealistic assumptions (see Wilhelm et al. (2008)). A alternative approach may be to implement PLCs on top of PRET machines, which are discussed by Zimmer et al. (2014).

Second, we must delay actuation based on results of a cycle until the minimum cycle time has elapsed. 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. 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 resulting simplicity in simulation and analysis compare to actual execution times.

More complex scenarios present additional challenges for virtual prototyping, but as demonstrated by PTIDES, these challenges can be overcome even in distributed systems, as shown by Eidson et al. (2012). Although PTIDES may not 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.

4. CONCLUSION

With the advent of Industry 4.0, industrial automation is facing conflicting demands of increasing complexity and safety requirements. At the same time, technology is offering more sophisticated timing-sensitive networks, open networks, multicore architectures and increasingly complex microprocessor architectures, all of which stress current PLC platforms. We believe it is time to reexamine these practices with an eye towards improving determinism, enabling virtual prototyping, leveraging multicore architectures and strengthening safety guarantees. We hope that this article will inspire future efforts towards the opportunities identified above.

REFERENCES

Benveniste, A. and Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), 1270–1282.

Berger, H. (2006). *Automating with SIMATIC: Controllers, Software, Programming, Data Communication, Operator Control, and Process Monitoring.* Publicis Corporate Publishing, 3rd edition.

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

Cengic, G., Ljungkrantz, O., and Akesson, K. (2006). Formal modeling of function block applications running in IEC 61499 execution runtime. In *11th IEEE International Conference on Emerging Technologies and Factory Automation.* doi:10.1109/ETFA.2006.355187.

Eidson, J., Lee, E.A., Matic, S., Seshia, S.A., and Zou, J. (2012). Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1), 45–59. doi:10.1109/JPROC.2011.2161237.

Eidson, J.C. (2006). *Measurement, Control, and Communication Using IEEE 1588.* Springer.

International Electrotechnical Commission (2017). *International Standard IEC 61131: Programmable Controllers.* IEC, 4.0 edition.

Jerad, C. and Lee, E.A. (2018). Deterministic timing for the industrial internet of things. In *IEEE Int. Conf. on Industrial Internet (ICII).* IEEE.

Kim, H. and Lee, E.A. (2017). Authentication and authorization for the Internet of Things. *IT Professional*, 19(5), 27–33. doi:10.1109/MITP.2017.3680960.

Lee, E.A. and Sirjan, M. (2018). What good are models? In *Formal Aspects of Component Software (FACS)*, volume LNCS 11222. Springer.

Lee, E.A. and Zheng, H. (2007). Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, 114 – 123. ACM. doi:10.1145/1289927.1289949.

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

Vyatkin, V. (2013). Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 9(3), 1234–1249.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., et al. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 1–53.

Wollschlaeger, M., Sauter, T., and Jasperneite, J. (2017). The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1), 17–27. doi:10.1109/MIE.2017.2649104.

Zimmer, M., Broman, D., Shaver, C., and Lee, E.A. (2014). FlexPRET: A processor platform for mixed-criticality systems. In *Real-Time and Embedded Technology and Application Symposium (RTAS).* URL `http://chess.eecs.berkeley.edu/pubs/1048.html`.