The Convergence of Bits and Atoms: Foundations, Verification, and Scale in Cyber-Physical Systems

TAREK ABDELZAHER, University of Illinois Urbana-Champaign, USA
EDWARD A. LEE, UC Berkeley, USA
SANJIT A. SESHIA, UC Berkeley, USA
WANG YI, Uppsala University, Sweden

Cyber-physical systems (CPS) couple computing and networking with physical dynamics, so that timing, interaction, and concurrency become essential features rather than performance side effects. Classical algorithmic abstractions—centered on terminating, time-agnostic input–output functions—are therefore insufficient for software choreographed with the physical world. This article surveys foundational and methodological advances that address that gap along four complementary lines. First, we argue for models of computation that treat time, nontermination, interaction, and deterministic concurrency as first-class concerns, including coordination languages and clock-synchronized distributed execution that manage consistency–timeliness tradeoffs. Second, we review design-for-predictability approaches—synchronous reactive design, logical execution time, and real-time scheduling—together with formal modeling and verification based on timed and hybrid automata, as exemplified by UPPAAL and related tools. Third, we examine connectivity and scale: macroprogramming, IoT middleware, decision-driven resource management, and emerging agentic frameworks for city-scale sense-act loops. Fourth, we discuss the two-way intersection of formal methods and AI: learning-based synthesis of models, specifications, and contracts for CPS verification, and the verified AI agenda for assuring AI-enabled physical systems. Together, these perspectives outline a path from conservative, hardware-frozen automation toward scalable autonomy with principled guarantees.

CCS Concepts: • **General and reference**;

Additional Key Words and Phrases: cyber-physical systems, models of computation, real-time systems, timed automata, formal verification, distributed systems, concurrency, Internet of Things, formal methods, verified AI

ACM Reference Format:
Tarek Abdelzaher, Edward A. Lee, Sanjit A. Seshia, and Wang Yi. 2026. The Convergence of Bits and Atoms: Foundations, Verification, and Scale in Cyber-Physical Systems. *ACM Trans. Cyber-Phys. Syst.* 0, 0, Article 0 (2026), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction: The CPS Paradigm Shift

In 2006, Helen Gill of the U.S. National Science Foundation coined the term "cyber-physical systems" (CPS) to describe systems that combine computing, networking, and physical dynamics. The term "cyber" originates from *cybernetics*, coined by Norbert Wiener in 1948, derived from the Greek word for helmsman or governor. Wiener pioneered technology for automatic feedback control, which serves as the ancestor to modern computer-based control logic. The agenda of CPS is to manage

Authors' Contact Information: Tarek Abdelzaher, zaher@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA; Edward A. Lee, eal@berkeley.edu, UC Berkeley, Berkeley, CA, USA; Sanjit A. Seshia, sseshia@berkeley.edu, UC Berkeley, Berkeley, CA, USA; Wang Yi, yi@it.uu.se, Uppsala University, Uppsala, Sweden.

ACM 2378-9638/2026/0-ART0
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the complexity created by conjoining the digital world with physical processes and to improve the engineering methods used to create such conjoined systems [71].

Consider a modern commercial airplane such as the Airbus A350 or the Boeing 787. These systems are software-intensive with hundreds of microprocessors managing functions that include translating pilot commands into rudder movements, controlling landing gear, managing cabin pressurization and airflow, managing electric power generation and distribution, and operating the passenger entertainment system. Such aircraft include complexities not present in an application that is more purely information processing, such as a spreadsheet handling accounting data. The latter only has to deal with digitally represented information.

A spreadsheet is an information-processing system that can operate almost entirely within the models and paradigms of computer science. An airplane design conjoins models of aeronautical, mechanical, electrical, and civil engineering with those of computer science. Despite the complexity and challenges of integrating so many engineering disciplines, Boeing and Airbus manage to make astonishingly safe and reliable airplanes. How? Today, their design processes and methods are extremely conservative and regulatory oversight is heavy. Many rules and processes must be followed to get an aircraft certified to carry civilian passengers.

A symptom is a story from an engineer who had worked on the Boeing 777, which was Boeing's first fly-by-wire airliner, meaning that pilot controls are mediated by a computer. The 777 first entered service in 1995. According to this engineer, as of the early 1990s, Boeing expected this model of aircraft to be in production for perhaps 50 years, to 2045, so they purchased a 50-year supply of the microprocessors for the flight control system. They expected to need to use the same physical microprocessors for the entire production run of the aircraft. Engineers at Airbus, which has been making fly-by-wire aircraft for longer than Boeing, confirm this story. They store the microprocessors in liquid nitrogen in an attempt to extend their shelf life.

Any particular silicon chip is unlikely to remain in production for more than a few years. It rapidly becomes obsolete under the pressure of technological advances. And when a fab is updated to leverage a new technology, it becomes impossible to produce an identical chip. But why does Boeing need the chip to be identical? A primary purpose of programming languages is to isolate software designs from changes in the hardware. The software for the 777 is mostly written in the C programming language. Shouldn't it be enough to verify that the code is correct, ideally using formal methods, and then use any chip that can correctly execute the code? This paper explains why this is not sufficient. The notion of correctness of the code needs to change to encompass the properties of the physical world with which the code will be choreographed.

2 The Foundations: Determinism and Models of Computation

Starting at the instruction set architecture (ISA) layer, and for nearly all abstraction layers above that, including the C programming language, what it means to *correctly* execute software has nothing to do with how long it takes to do anything. Timing of actions is not part of the programming paradigms used today. The prevailing paradigm today instead relies on the notion of an algorithm. Retrofitting this paradigm to encompass temporal dynamics turns out to be nontrivial.

As with most successful paradigm shifts [65], we do not expect to discard algorithms. Far from it. But we need to augment them with models of temporal dynamics, with deterministic concurrency, and with time-sensitive distributed computing capabilities. These enhancements are needed to ensure that the "map" (the model) accurately reflects the "territory" (the conjunction of the cyber and the physical).

2.1 Algorithms and Time

A traditional algorithm is a step-by-step calculation procedure, a recipe. It is an abstraction of what a classical von Neumann computer actually does. In this abstraction, the time taken by a step is not part of the model. In part because of the centrality of algorithms, relatively little in the theory of computer science is about when, in time, a result is produced or an action is taken.

The subfield of complexity theory [11, 122] is related to timing. However, it is about how the number of steps varies with the problem size, not about when in time an interaction with the world outside the program occurs. In complexity theory, for example, an algorithm where the number of steps is an exponential function of the problem size may be less useful than an algorithm where the number of steps is a quadratic function. Whether a result is produced in ten milliseconds or one hour is not part of the theory. However, for the A350 and 787, *when* a computation completes has an enormous impact on the behavior of the aircraft.

The subfield of real-time systems is concerned with the design of timing-sensitive software [118]. It finds ways to specify timing requirements and develops techniques to schedule tasks to ensure that these timing requirements are met. Hard real-time systems [24] are a subclass where failing to meet the timing requirements is considered a critical failure. Sophisticated methods have been developed, but all of them are hobbled by the lack of temporal semantics in programming models. Most of these theories assume that execution times for software are known, or at least bounded, but this assumption is not compatible with the programming abstractions and hardware designs that are used in practice. Execution times can only be known or bounded on particular pieces of hardware operating in a particular context, sometimes even depending on temperature and battery charge levels. Hence the Boeing and Airbus dilemma; they can only be sure of the behavior if the hardware remains the same over the 50-year production run.

Another relevant subfield of CS is concerned with temporal logics [46, 68, 104]. These logics enable reasoning about behaviors of programs over time. The core theories, however, do not treat time the same way that a mechanical engineer would, for example. They talk about "eventually" and "always" rather than directly about temporal dynamics. Extensions known as metric temporal logics [34, 46] add a notion of time intervals, allowing logical predicates like "within 10 ms." These offer a significant step towards useful modeling for CPS, but still face a mismatch of abstractions with the underlying, time-agnostic algorithms.

In a flight control system, the software is directly controlling physical actuators, and in the physical world, timing is critical. Just about every model that an aeronautical, mechanical, or electrical engineer uses has timing as a central property of the model. The paradigms used by these engineers are incommensurable with the paradigms used by computer scientists. As a consequence, a programming language fails to provide an adequate abstraction.

Even with a fixed physical instance of a microprocessor, the fact that timing is irrelevant to correctness from the ISA up means that in software design, aircraft manufacturers cannot use most of the innovations of computer science from the last 50 years. Object-oriented languages such as Java and C++, for example, are often avoided in safety-critical software. Even interrupts, the standard way that all modern microprocessors get data in and send data out, are avoided. To be sure, interrupts create many subtle software problems. As far back as 1972, Edsger Dijkstra lamented,

> [I]n one or two respects modern machinery is basically more difficult to handle than the old machinery. Firstly, we have got the interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton. This has been a dramatic change, and many a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature. [32]

Despite this lament, to this day, interrupts remain the primary method for I/O and are central to every modern operating system design, including Windows, macOS, Linux, Android, and iOS.

Aircraft manufacturers are not alone in facing this problem. Modern cars are mostly drive-by-wire, where the driver commands (pushing on the accelerator and brake pedals and turning the steering wheel) are mediated by a computer before going to the wheels or engine. Automotive designers face the same problems, but with fewer regulatory constraints. Their problems only get worse as automation increases (lane keeping, automated accident prevention, and fully automated driving). The introduction of artificial intelligence (AI) further complicates this story, as discussed in Section 2.4 below.

It does not stop there. Any modern factory is computer controlled and is similarly a safety-critical system where the timing of actions is essential to safety. Trains, ventilation, lighting, fire mitigation systems in buildings, electric power grids, water distribution systems, and communication systems are all computer controlled.

One effort to address the mismatch in abstractions is to endow software with precision timing capabilities at the microarchitecture and ISA levels, creating what are known as PRET machines [39, 64, 85, 108, 136, 144]. Some of these machines are even capable of handling interrupts with zero disruption of real-time threads. Even though it has been shown that there is no loss of performance for a large class of workloads [79], these architectures remain a niche research topic. They are very much out of the mainstream, perhaps because of the lack of well-developed overlaying software abstractions. There is promising work in this direction, however [8, 86, 98].

Even though the foundations of CS are about algorithmic processing of digital information, any practical information processing application that interacts with the world outside the computer has a CPS element. Even a spreadsheet has timing concerns; human users will get impatient if updates take too long. But here, timing is a performance metric, whereas in many CPS applications, timing becomes a correctness criterion, every bit as important as whether arithmetic is done correctly. If the timing is off, the implementation is flawed.

The discipline of cyber-physical systems is specifically concerned with the conjoining of the physical world, with all its time-sensitive dynamics, with the world of algorithms. It works on modifying both sides, admitting computational intelligence to the physical side and temporal dynamics to the cyber side.

2.2 Nontermination

In the traditional theory of computer science, an important feature of an algorithm is that it reaches a conclusion. That is, it halts, giving a final answer. The purpose for an algorithm is to determine that answer. The physical side of a CPS does not halt (or even pause). Hence, the cyber side should not halt either.

Modern computers routinely run programs that are designed to not halt, such as an operating system. Manna and Pnueli [93] show that nonterminating processes like operating systems cannot be modeled by algorithms. These programs do occasionally halt, but we describe such halting as a "crash" or a "deadlock," making it quite clear that this was not intended. A program that does not halt does not realize an effectively computable function and hence is not performing a Church-Turing computation. Various pieces of what it does need to reach conclusions in a bounded number of steps, and hence do fall within this theory, but, holistically, the nonterminating behavior of the overall program is not well modeled by a classical algorithm. And nearly all programs for CPS applications are designed to not halt unless explicitly asked to.

2.3 Interaction

Related to timing and nontermination, another property that is central to CPS but largely missing from the classical algorithmic paradigm of computing is interaction. An interactive program is one that produces *partial* output given *partial* input rather than giving a single final output given the totality of its inputs.

Interaction enables systems where an output may depend on previous outputs. A program may probe its environment, testing it and measuring its response before producing further output. Feedback control systems, which are common in CPS, have this character. The control software issues a command to the environment, measures its response, and adjusts its further commands to improve the response. It is possible to learn things about a system by interacting with it that cannot be learned through passive observation [72].

Wegner [131] argues that interactive programs can do more than algorithms. In a summary of a panel held at the Workshop on Foundations of Interactive Computation in Edinburgh in 2005, Wegner et al. [132] conclude that "today's computation applications, such as web services, intelligent agents, operating systems, and graphical user interfaces, cannot be modeled by Turing machines." They say that it is a "Turing Thesis myth" that a Turing machine can do anything a computer can do. Hewitt [54] also claims that timing of the interactions of programs can affect behaviors in ways that make the computations richer than what Turing machines can accomplish.

Milner [94, 95] provides one possible formal foundation for this observation. He shows that two systems that are accurately modeled by identical input-output functions may nevertheless be distinguishable through interaction. He developed a stronger notion of equivalence between systems that he called bisimulation. Two systems that are bisimilar are indistinguishable even through interaction.

If a program is interacting with the physical world (i.e., it is the cyber part of a CPS), then the timing of the actions of the program will affect the overall behavior of the system, including the program's own future outputs. Such a program is clearly *not* a Turing computation because Turing's model includes no notion of time, and the timing of outputs (or inputs) has no effect on the computation.

An interactive program may be interacting with another interactive program. These two programs may even be executing on the same machine if the machine is capable of multitasking, as most modern computers are. Such a pair of programs is said to be *concurrent*. Again, the timing of actions may affect the overall behavior, so Turing's model requires some extension to include such systems. Milner, in his Turing Award lecture [95], observed that concurrent programs cannot be modeled simply as functions from inputs to outputs, as (halting) Turing machines can be. Their chunks can be modeled as such functions but not their overall behavior.

2.4 Artificial Intelligence

Today's large-language models (LLMs) are realized on computers, albeit ones bearing little resemblance to Turing machines. At their lowest level, computer programs specify algorithms operating on formal symbols. However, deep neural networks (DNNs), key components in an LLM, exhibit behaviors that are not usefully explained in terms of the operations of these algorithms [73]. An LLM is implemented on computers that perform billions of logic and arithmetic operations per second, but even a detailed knowledge of those operations gives little insight into the behaviors of the DNNs [22].

LLMs are fundamentally based on prediction. Abelson and Sussman, in the preface to their classic introduction to computer science [3], describe the computer revolution as a "procedural epistemology," knowledge through procedure. We have entered a second computer revolution,

where prediction trumps procedure. Knowledge through prediction is what today's AIs exhibit. Logic, mathematics, and computer programming are things they do rather than what they are, just like humans.

Kahneman identifies two distinct human styles of thinking, a fast style (System 1) and a slow style (System 2) [62]. The slow style is capable of algorithmic reasoning, but the fast style, which is more intuitive, is responsible for many of the decisions humans make. It turns out that an inference step in an LLM more closely resembles System 1 than System 2 [74]. Even though they are realized on computers, how they reach decisions is not well modeled by algorithmic reasoning. Practitioners are learning how to teach the machines to make more use of algorithmic reasoning, for example using CoT (chain of thought) strategies [133], where the machines perform sequences of inferences rather than one shot. When asked to solve algebra problems, for example, the machines do better if they are asked to explicitly give a sequence of reasoning steps [22].

While CoT resembles an algorithmic process, it bears an odd relationship to the procedural epistemology enabled by the low-level operations of a computer. First, CoT is very limited. The machines make more errors when the number of steps gets large [120]. There is no such issue with the low-level operations, where billions of operations per second do not stump the machines. It seems that, like humans, AIs have bounded rationality [73, 121], even if their underlying machines do not.

Cyber-physical systems add a distinctly new element by enabling AI interaction with the physical world. Self-driving cars, drones, and humanoid robots already integrate AIs in tight feedback loops with their physical environment. Other CPS applications, such as industrial robots, automated trains, and the electric power grid, are carefully experimenting with and more slowly acquiring AI capabilities. A major challenge is that ensuring safety in such systems is more challenging with AI integration because their System 1-like processes are less predictable than algorithmic reasoning [75].

2.5 Distributed CPS

Distributed systems, where components communicate over networks, have more intrinsic uncertainties. As a consequence, many designers throw up their hands and decide that they do not need deterministic models, opting instead for easy-to-use publish-and-subscribe middleware, actor networks, and remote procedure calls. But these choices come with considerable costs. Because the underlying semantic models are nondeterministic, systems do not define a single correct response to a particular input stimulus. This makes it much more difficult to form consistent views of system state across a distributed system, to detect faults at runtime, and to design regression tests that protect against errors at design time. Is an observed behavior one of the many allowed by the semantic model? Or is it the result of a fault in the system? Moreover, anomalies in such nondeterministic semantic models are often rare and difficult to reproduce, which means that systems can perform well during extensive testing and then fail in the field.

A key challenge in distributed systems is to form a consistent view of the state of the system across distributed components. There are many applications that require such consistency. For example, autonomous vehicles should not enter an intersection without a consistent view across vehicles of the state of the intersection.

Loosely, consistency is agreement on shared information, where the information can change over time. In certain cases, if the information changes can be modeled by a monotonic function in some partial order, then *eventual* consistency can be achieved without coordination [67]. However, "eventual" may not be good enough. If timing matters, then consistency requires coordination [82]. Moreover, fundamental limits show that, as network latencies increase, the time taken to form a consistent view increases [77].

Most frameworks for coordinating concurrent and distributed software components provide little or no help towards ensuring consistency or managing the timing costs. Publish-and-subscribe frameworks (such as ROS 2 and MQTT), actor frameworks (such as in Erlang, Akka, Orleans, and CAF), and remote procedure call frameworks (such as gRPC and Apache Thrift) are particularly weak in this regard. There are better alternatives.

Given now that coordination is required, what coordination mechanism should we use? One strategy is to centralize shared information, but this has many drawbacks. For one, it creates a single point of failure, where failure of the central store can doom a system. Moreover, it requires very careful design, often well beyond the skill set of system engineers. Given that it will take time to access central information, for example, how can a component be sure the information is still valid when it receives it?

One promising and potentially ubiquitous coordination mechanism, clock synchronization [60], can provide a foundation for distributed systems with consistency and timing guarantees, up to fundamental limits. Clock synchronization with bounded error provides a foundation for time-stamping messages and providing a useful global semantic ordering that can guarantee consistency.

The reactor model of computation and the Lingua Franca (LF) implementation in a coordination language [87] make timestamps a central feature, providing a logical timeline [86] that enables deterministic concurrency. Concurrent components exchange timestamped messages, and every component processes events in timestamp order.

In principle, the deterministic concurrency of reactors can be achieved without coordination if timing does not matter and interaction is not needed [82]. But for CPS, timing and interaction are always needed, so this is not useful. Some level of coordination becomes essential.

When LF programs are distributed across networks, two distinct coordination mechanisms are provided in the open-source code base [13]. A *centralized coordinator* mediates all interactions in order to ensure deterministic semantics without relying on clock synchronization or bounded communication latencies. A *decentralized coordinator*, in contrast, relies on clock synchronization and enables tradeoffs between ensured consistency and timeliness [76].

The decentralized coordinator in LF is much more interesting for CPS applications. It has no single point of failure, no centralized bottleneck, and can provide a full range of consistency strategies from fully guaranteed (at the fundamentally unavoidable price of potentially unbounded delays) to guaranteed under the assumption of bounded network latencies. With the weaker guarantees, the LF mechanisms also provide hooks for application-specific fault handlers to be invoked when assumptions about network latency bounds are violated or when components fail.

3 Design and Verification of Real-Time Systems

Given that, in CPS, time is not merely a performance metric but a fundamental dimension of correctness, we face a *predictability challenge* with two complementary aspects: *timing correctness*, which requires actions to occur within their specified temporal constraints, and *functional correctness in the presence of time*, which requires the system's logical behavior to remain correct despite implementation-level variations such as timing nondeterminism, resource contention, and scheduling. A software component can be considered predictable when its behavior is well-defined as a function: for given inputs, both the resulting output and its production time can be determined. Such temporal and functional repeatability is essential for reliable CPS [80].

Two complementary approaches have been developed to achieve predictability. The first is *predictable-by-design*, where correctness is guaranteed by the methodology itself [53]. The second is *modeling and verification*, where a formal model of the system is constructed and mathematical techniques are used to check whether it satisfies the correctness requirements [5, 6, 142].

3.1 Predictability by Design

This strategy has been pursued along at least three lines: synchronous design [18], logical-time models such as Logical Execution Time (LET) [52] and reactors [88], and physical-time real-time systems including the classic approach based on real-time scheduling [25, 84] and asynchronous design aimed at determinism, modularity and updatability [140, 141]. These approaches differ in their assumptions about when computation occurs and how components interact, trading off determinism, implementation flexibility, and verifiability.

Synchronous Design and Reactive Systems. In the synchronous design paradigm [18, 48], a system is decomposed into components that execute in discrete logical ticks. All components share the same global tick, which defines the execution rate. At each tick, every component reads its inputs, computes its reaction according to its specified function, and writes its outputs, with reactions ordered by communication structure and data dependencies. The *synchronous hypothesis* assumes each reaction takes zero logical time [18], so components reacting at the same tick are conceptually simultaneous. This enables a global logical clock that orders all interactions independently of physical execution times, provided that all reactions complete before the next tick begins. The paradigm has been realized in several influential tools and languages. *Lustre* and its industrial successor *SCADE* are dataflow languages that execute synchronously at each tick; SCADE has become a standard for safety-critical avionics applications [48]. *Esterel* is an imperative synchronous language where reactions are logically instantaneous [20]. *Simulink* is the most widely used tool for model-based design of embedded control systems, supporting hierarchical block diagrams with synchronous semantics.

The global tick that defines the execution rate enables determinism but also creates tight coupling between components and their schedules. Since all components share the same tick, their rates must be harmonized. In a single-rate implementation, all components execute at the same rate; adding a faster component requires changing the global tick rate to the greatest common divisor of all rates, potentially affecting unrelated components [52]. Multi-rate models allow different rates, but adding a new rate still requires recomputing the overall schedule. This makes synchronous systems difficult to modify, motivating an abstraction where components have more independent timing while preserving deterministic interactions.

Logical Time and Distributed Systems. A more general concept is *logical time*: an abstraction where time is ordered but not necessarily tied to physical time. Unlike the synchronous paradigm, where all components share a common tick, logical time allows deterministic ordering of events without requiring global synchrony. The *Logical Execution Time (LET)* paradigm, introduced in Giotto [52], makes this separation explicit. A task has a logical release time for sampling inputs and a logical completion time for making outputs visible; computation may occur at any physical time between these points, as long as it finishes before completion. Two implementations with different execution times can thus exhibit identical external behavior. LET provides a contract at the task interface: the environment observes when data is logically consumed and produced, not the schedule used to compute it. A synchronous reaction is the limiting case with zero logical execution time. The trade-off is that LET itself does not guarantee that the physical implementation can meet this interval; scheduling and execution-time analysis must establish that separately.

LF [87] generalizes LET to concurrent and distributed reactive systems [78]. While LET specifies a fixed logical delay, LF decouples the logical timeline from physical time except at explicit interactions with sensors, actuators, and timers. LF represents interactions as timestamped events: a reactor reacts when an event arrives, and the timestamp determines its logical ordering, so components execute independently while maintaining deterministic interactions. For distributed systems, where

execution times and communication delays prevent a common instantaneous notion of time, LF provides a shared logical ordering that preserves deterministic behavior. As with LET, logical determinism does not guarantee physical deadlines; that guarantee is left to the execution platform and scheduling analysis [78].

Physical Time and Real-Time Systems. When computation time is directly relevant to the physical process, timing requirements must be met in physical time as measured by a clock. This is the domain of real-time systems, where components with heterogeneous timing requirements are modeled as tasks with individual timing and resource constraints. Such heterogeneity is common in practice: a braking function may require millisecond response, while a temperature monitor may execute once per second. The classical results of Liu and Layland [84] established fundamental properties of periodic task systems and showed the optimality of Earliest Deadline First (EDF) for preemptive uniprocessor models. Later work extended these results to sporadic arrivals, varying execution requirements, mode changes, and more complex dependencies [96, 124]. For uniprocessor systems, the demand-bound function provides a general feasibility test by determining whether processor demand can be accommodated in every relevant interval [12]. For multiprocessor platforms, however, while solutions exist for specific scheduling policies and task models, a unified foundation comparable to the uniprocessor theory remains an open challenge.

The explicit treatment of real-time tasks provides considerable implementation flexibility: each task can be scheduled independently according to its own timing constraints, rather than being forced into a common execution rate. However, meeting individual timing constraints does not by itself ensure that the observable behavior of interacting tasks is deterministic, since tasks may observe shared data at different physical times. The challenge is to combine implementation flexibility with the deterministic observable behavior ensured by logical-time models.

Deterministic Real-Time Systems and Updatability. Synchronous systems offer determinism but are notoriously hard to update. A central challenge is therefore to combine the implementation flexibility of real-time systems with deterministic behavior, while also supporting composability and incremental, safe system evolution. Asynchronous design as proposed in MIMOS addresses this challenge, targeting three key properties: determinism, modularity, and updatability.

MIMOS achieves determinism by decoupling task execution from communication through asynchronous FIFO queues and registers, while preserving LET-style read and write semantics [140, 141]. Tasks may execute at different rates and communicate via FIFO queues, which preserve message order and provide buffering, or via registers, which retain only the most recent value. As in LET, each task reads its inputs at release and writes its outputs at its deadline, with computation in between. Provided all timing constraints are met, variations in execution caused by platform non-determinism, workload changes or scheduling policies do not affect observable behavior, thereby guaranteeing determinism.

Modularity and updatability follow naturally from this construction. A component interacts with the rest of the system only through its well-defined timing and communication interfaces. Adding a new component or replacing an existing one does not affect the execution of other components, and requires neither system-wide redesign nor revalidation, as long as the extended system remains schedulable. Moreover, these explicit interfaces provide a basis for analyzing the effect of an update before deployment [141].

3.2 Modeling and Verification

While predictable-by-design approaches aim to build predictability into systems from the outset, they cannot eliminate the need for rigorous verification. Modeling and verification therefore provide

a complementary framework for checking whether a design satisfies its temporal and functional requirements.

Early tools for automated analysis of timed systems included Kronos [21], developed at Verimag for model checking timed automata [5], and TAU[1], developed at Aalborg University for checking bisimulation equivalence of timed CCS [139]. Both relied on region-graph construction [5] to establish decidability of reachability for timed automata, but were limited by the combinatorial explosion of region graphs and the product composition of concurrent components. In parallel, UPPAAL [70] was initiated at Uppsala for on-the-fly symbolic verification of communicating timed automata [142]. Rather than explicitly constructing region graphs, it represented timing constraints symbolically and explored the state space on the fly, following an approach previously adopted for untimed systems by the SPIN model checker [55]. This avoided pre-constructing the full product of concurrent components. The initial Prolog prototype was presented at FORTE in 1994 [142]; in 1995, the Aalborg team joined the effort, the tool was named UPPAAL, and its first C-based implementation was released and demonstrated at TACAS [16].

Timed Automata and UPPAAL. A timed automaton, introduced by Alur and Dill, is a finite-state automaton equipped with clocks that increase synchronously and can be reset to zero [5]. It consists of locations, one of which is initial, and transitions labeled with actions, guards, and clock resets. Guards constrain clock values, while resets allow elapsed time to be measured from a new reference point. Each location may also have an invariant that must remain true while the automaton resides there, thereby forcing it to leave before a specified time. These mechanisms allow properties such as "the message arrives within 3 to 5 time units" or "the controller must respond within 10 milliseconds" to be modeled directly. In the original theory, progress properties were specified using Büchi accepting conditions; UPPAAL instead uses location invariants and related constructs that provide an intuitive basis for safety verification while preserving the relevant expressiveness [17, 70].

The fundamental verification problem for timed automata is reachability: whether an unsafe state can be reached from the initial state. Although clocks range over an infinite continuous domain, reachability remains decidable because clock valuations can be partitioned into finitely many equivalent regions [5]. Two valuations are equivalent when they agree on the integer parts of clock values, which clocks are zero, and the ordering of their fractional parts. The resulting finite region automaton preserves reachability, but its size is exponential in the number of clocks and the magnitudes of the constants. More complex problems, such as universality and language inclusion, are undecidable for nondeterministic timed automata, although they become decidable for deterministic subclasses such as event-recording automata [4].

UPPAAL describes systems as networks of timed automata extended with discrete data variables. Automata communicate through synchronization channels, while integer variables represent shared discrete state. The language also provides constructs for time-critical interactions. *Committed locations* prevent time from passing and restrict interleaving until the corresponding sequence of committed transitions is completed, while *urgent channels* prevent time from elapsing when the corresponding synchronization is enabled. These constructs allow common forms of atomic and time-critical behavior to be represented directly. Properties are specified separately from the system model using UPPAAL's query language, a fragment of CTL extended with timing constraints [70]. Reachability and safety properties can be expressed directly, while more complex timing requirements can be captured using temporal queries and observer automata. An observer can monitor the system and enter an error location when a specified behavior or timing constraint is

[1]The TAU tool was never publicly released or published.

violated. This separation of models and properties supports incremental modeling and verification [83].

Symbolic and On-the-Fly Verification. UPPAAL makes reachability analysis of timed automata practical by combining symbolic clock-constraint representations with on-the-fly state-space exploration [17]. It represents sets of clock valuations as *zones*: convex sets described by constraints on clock values and their differences. Zones can be represented efficiently using *Difference Bound Matrices (DBMs)*, which support the main verification operations, including adding guards, letting time elapse, resetting clocks, and normalizing constraints. This avoids enumerating individual regions and substantially reduces the explored state space. On-the-fly verification further avoids constructing the complete state space before checking a property. Instead, the verifier explores the symbolic state space as needed, which is particularly important for concurrent timed automata, where explicit product construction can cause rapid state-space growth. Exploration can terminate as soon as a property violation is found, while irrelevant portions of the state space remain unexplored [142].

The combination of symbolic zones, on-the-fly exploration, and state-space reduction was central to making UPPAAL applicable beyond small examples. Subsequent work introduced more compact data structures and additional reduction techniques to reduce memory consumption and verification time [69]. The practical applicability of these techniques was demonstrated through case studies including the Philips Audio Protocol [15], an audio and video protocol from Bang & Olufsen [49], the SAAB gear controller [83], and the ABB field bus protocol [30]. These studies showed that symbolic timed-automata verification could be applied to realistic communication protocols and embedded controllers and could reveal timing and concurrency errors that are difficult to detect through testing alone.

Beyond Timed Automata and Hybrid Systems. UPPAAL has been extended beyond its core timed-automata framework. *TIMES* [7] adds real-time scheduling and schedulability analysis, including synthesis of executable code from timed-automata models. Other extensions support *priced automata* for cost-optimal reachability [14] and *timed games* for controller synthesis [26]. Together, these extensions broaden timed automata from verification to scheduling, optimization, and synthesis. For CPS involving continuous dynamics, *hybrid automata* extend timed automata with continuous variables whose evolution is governed by differential equations [6]. A hybrid automaton combines discrete control modes with continuous flows, invariants, and transitions, allowing physical processes such as vehicle motion and temperature evolution to be modeled alongside discrete control behavior. The *HyTech* tool, developed by Henzinger and colleagues, was among the first tools for automatic analysis of hybrid systems [51]. It was applied to hybrid-system benchmarks including railroad gate and temperature-control systems. Despite the high complexity of even decidable subclasses, HyTech demonstrated the feasibility of automated verification for practical hybrid-system models.

4 Connectivity and Scale: The Networked Sense-Act Loop

As CPS scales to city-scale (Smart Cities and Intelligent Transportation), the focus shifts to middleware and networked loops. This perspective emphasizes the importance of managing data flow between thousands of sensors and actuators while maintaining real-time guarantees. In particular, higher-level abstractions are needed to allow designers to specify aggregate system behavior and purpose while letting the underlying real-time resource management solutions decompose those aggregate specifications into algorithms that prioritize tasks, schedule communication, and distribute load in a manner that meets mission requirements and time constraints. Several examples follow.

4.1 Macroprogramming

Early examples of high-level programming abstractions for distributed CPS were developed in the sensor network community that recognized that programming large-scale wireless sensor networks (WSNs) at the level of individual nodes was becoming increasingly impractical. Networks consisting of hundreds or thousands of resource-constrained devices required programming models that abstracted away low-level concerns such as message passing, routing, synchronization, and node failures. This realization led to the emergence of *macroprogramming* [125], whose central objective was to allow developers to specify system behavior from a global, application-centric perspective while automatically compiling or distributing that specification into node-level implementations. By 2010, macroprogramming became one of the dominant research directions in sensor network systems, producing a diverse collection of programming models that introduced new abstractions based on nodes [47], regions/neighborhoods [99, 134, 135], databases [90], environmental objects [2, 89], declarative languages [28], functional programming [91, 99, 100], and distributed dataflow [99]. Although differing substantially in their implementation mechanisms, these systems shared the common goal of elevating the programming abstraction from individual devices to the collective behavior of the network.

Early work was motivated by the observation that many sensing applications were fundamentally interested in *physical phenomena* rather than individual sensors. Environmental monitoring [102], structural health monitoring [101], habitat observation [92], and surveillance [50] all required reasoning about aggregate measurements over geographical regions instead of interacting directly with particular nodes. Macroprogramming therefore sought to make sensors largely invisible to application developers by introducing abstractions representing logical collections of devices, spatial regions, or distributed datasets. The runtime system became responsible for mapping these abstractions onto the dynamically changing physical network.

An example was to formulate the sensor network as a *distributed database*, such as the TinyDB system [90]. It took advantage of the observation that many sensing tasks could be naturally expressed using declarative SQL-like queries over continuously generated sensor data. TinyDB established several principles that would strongly influence later macroprogramming systems, including separating application intent from execution strategy, exploiting in-network processing to reduce communication costs, and optimizing resource consumption through automatic query planning.

A second major direction focused on *spatial programming abstractions*, recognizing that sensor network applications naturally reason about geographic regions and neighborhoods. One of the most influential examples was *Abstract Regions* [134], which proposed treating spatial regions as first-class programming objects. Applications manipulated regions instead of enumerating individual nodes, allowing operations such as sensing, communication, and actuation to be expressed over dynamically defined geographical collections. The runtime system was responsible for determining which physical nodes belonged to each region and maintaining consistency as topology evolved. This abstraction significantly simplified programming for applications involving environmental monitoring, target tracking, and spatial coordination.

Closely related work extended these ideas through logical neighborhood abstractions. *Hood* [135] provided programmers with a view of neighboring nodes as shared data structures rather than communication endpoints. Instead of explicitly exchanging messages, applications read neighborhood state through a distributed shared-memory abstraction maintained by the runtime. Similarly, *Kairos* [47] introduced programming constructs allowing developers to describe collective operations over dynamically changing sets of nodes while leaving communication scheduling and synchronization to the underlying runtime.

Another influential family of systems adopted *functional programming* principles. *Regiment* [99] became one of the best-known functional macroprogramming languages for sensor networks. It viewed the network as a collection of distributed data streams and introduced stream-processing operators inspired by functional programming languages. Computation was expressed as transformations over spatially distributed streams of sensor data, while a compiler automatically partitioned programs into localized node-level computations connected by communication channels.

Environmentally immersive programming [2, 89] was another paradigm that exported objects in the environment as the primary abstraction, allowing methods and data (or state) to be associated with such objects that would manipulate them via sensing or actuation. The run-time system would take care of object creation to maintain a one-to-one correspondence between the current physical state and its digital representation, and select the sensors and actuators necessary to implement the specified operations on the specified objects.

Several macroprogramming systems also explored rule-based and event-driven programming models [36, 37, 126]. Rather than explicitly controlling execution, developers specified reactions to events or logical conditions. Runtime systems monitored distributed state and automatically triggered computations when predicates became satisfied. These approaches reduced programming complexity for event detection, alarm generation, and context-aware applications while allowing implementations to optimize monitoring overhead dynamically.

Despite differences in programming style, several common themes unified nearly all macroprogramming research. First, virtually every system attempted to *decouple application logic from network implementation*. Programmers expressed *what* computation should be performed, while the runtime determined *how* it should execute across distributed nodes. Second, most systems emphasized *automatic optimization*, including communication reduction through in-network aggregation, intelligent operator placement, adaptive sampling, and energy-aware scheduling. Third, nearly all approaches relied upon *logical abstractions*, replacing explicit node identifiers with regions, streams, databases, neighborhoods, or collections that better reflected application semantics.

Although macroprogramming achieved considerable conceptual success, widespread adoption proved limited. Sensor network deployments became increasingly heterogeneous, integrating mobile devices, embedded processors, cloud services, and Internet connectivity rather than consisting solely of homogeneous motes. Moreover, early systems did not explicitly address time constraints thereby limiting their scope to softer CPS applications.

Nevertheless, the intellectual legacy of macroprogramming has proven remarkably enduring. Many abstractions introduced during this period, such as declarative data acquisition, distributed dataflow, logical neighborhoods, spatial collections, region-based computation, automatic operator placement, and compiler-driven resource optimization, have reappeared in modern edge computing, fog computing, stream processing, Internet-of-Things middleware, and distributed AI systems. Contemporary distributed sensing systems increasingly employ semantic data abstractions, declarative task specifications, automatic runtime optimization, and adaptive execution strategies that closely resemble the goals originally articulated by the macroprogramming community.

4.2 IoT Middleware

More contemporary IoT middleware [107] has evolved into a comprehensive *resource management layer* responsible for orchestrating computation, storage, networking, sensing, and AI inference across geographically distributed platforms while satisfying application requirements for latency, reliability, scalability, privacy, and energy efficiency.

A central responsibility of modern IoT middleware is *resource abstraction*. Instead of exposing individual devices, processors, sensors, and communication links directly to applications, middleware presents virtualized resources through uniform programming interfaces. Devices are represented as

services or digital resources that can be dynamically discovered, allocated, monitored, and migrated. This abstraction allows applications to remain independent of the underlying hardware while enabling the middleware to optimize placement and execution as resource availability changes. Virtualization technologies [19], lightweight containers [106], and microservice architectures [97] have become the dominant mechanisms for implementing these abstractions, allowing software components to be deployed seamlessly across cloud and edge infrastructures.

Another major function is *distributed task orchestration*. IoT applications are increasingly composed of interconnected processing pipelines that span multiple computational tiers. Middleware frameworks determine where computation should execute by considering processor utilization, network latency, available memory, energy consumption, and application deadlines. Dynamic task placement and migration algorithms continuously adapt to changing workloads, mobility, and failures. Rather than statically assigning functionality to devices, modern systems support runtime relocation of services to improve responsiveness or reduce communication overhead. Platforms such as Kubernetes [23], KubeEdge [137], OpenYurt [103], EdgeX Foundry [38], Eclipse ioFog [56], and AWS IoT Greengrass [66] exemplify this trend by providing container orchestration and lifecycle management across cloud-edge hierarchies.

Resource management middleware also performs *data management and communication optimization*. Since communication often dominates energy consumption and latency in IoT deployments, middleware determines how sensor data should be collected, filtered, aggregated, compressed, cached, and disseminated. Publish-subscribe systems such as MQTT brokers [123], DDS middleware [44], and Apache Kafka [45] decouple data producers from consumers while supporting scalable event dissemination. Edge caching and stream processing reduce redundant communication with cloud services by performing filtering and aggregation close to data sources. Modern middleware increasingly incorporates semantic metadata and context-awareness to optimize data movement according to application requirements rather than simply forwarding all available observations.

An increasingly important responsibility is the management of *AI and machine learning workloads*. Many IoT applications now execute neural network inference at the edge to satisfy stringent latency and privacy requirements. Middleware platforms allocate accelerators such as GPUs, NPUs, and TPUs, schedule inference pipelines, distribute models across heterogeneous devices, and coordinate collaborative execution between cloud and edge. Resource managers must balance competing objectives including inference latency, model accuracy, communication cost, and energy consumption. Research challenges include model partitioning, adaptive offloading, federated learning, and continuous deployment of updated models across distributed edge infrastructures.

4.3 Decision-Driven Execution

Many CPS applications have strong time constraints. General IoT frameworks are not usually designed with such constraints in mind. In contrast, an example IoT framework that specifically addresses real-time scheduling is decision-driven execution [1]. This paradigm fundamentally reorients resource management around the information requirements of decision-making. Instead of treating sensing, communication, storage, and scheduling as independent system services, the framework exposes the logical structure of application decisions to the runtime system, enabling the infrastructure to optimize data acquisition, caching, communication, and scheduling jointly for decision quality, timeliness, and resource efficiency. The paradigm is designed for data-intensive systems. Many IoT and cyber-physical applications exist primarily to support operational decisions based on real-time data. Traditional resource management allocates computation and communication resources according to application requests without understanding why particular data are needed. Consequently, systems retrieve the requested data even though some of it may ultimately

prove unnecessary. Decision-driven execution instead exports application decision logic to the system layer, allowing the runtime to understand which information is essential for evaluating alternative courses of action for the purposes of decision-making and to allocate resources accordingly. Resource consumption therefore becomes directly tied to the process of selecting a course of action among competing alternatives, rather than simply executing application code.

The central computational abstraction in decision-driven execution is the *decision query*. Rather than requesting individual data objects, an application submits a logical expression describing the predicates required to determine whether alternative actions are feasible. Decisions are modeled as Boolean expressions over predicates representing conditions in the physical world. Evidence collected from sensors is used to evaluate these predicates, and once sufficient predicates have been resolved to identify a viable course of action, additional data collection can terminate. This changes the interface between applications and resource management from specifying *what* information should be retrieved to specifying *why* it is needed. The resulting semantic information enables the runtime system to optimize sensing, communication, and storage based on the structure of the decision itself.

A major distinction of this paradigm lies in redefining resource management as an optimization problem centered on minimizing the cost of reaching correct decisions. Because the runtime knows the logical dependencies among predicates, it can exploit opportunities for *short-circuit evaluation* analogous to compiler optimization of Boolean expressions. For conjunctive conditions, predicates most likely to invalidate an alternative relative to their retrieval cost are evaluated first, whereas for disjunctive structures, predicates most likely to establish a viable alternative are prioritized. Consequently, communication resources are allocated not according to static priorities but according to their expected contribution toward resolving decisions.

From a scheduling perspective, the framework formulates *decision-driven scheduling* [63], which extends conventional real-time scheduling by simultaneously considering computational deadlines and physical data validity. Decision-driven scheduling recognizes that sensed information continuously ages because the physical world changes over time. Schedules must therefore satisfy both decision deadlines and freshness constraints ensuring that all evidence remains valid when decisions are made. These coupled cyber-physical timing constraints produce fundamentally different scheduling problems in which execution order depends jointly on resource contention and physical dynamics. Optimality results can be computed for simple versions of the problem. For example, for a single bottleneck and independent queries, the optimal strategy retrieves evidence with the longest validity intervals first, leading to the *Least Volatile First (LVF)* policy.

4.4 Agentic Frameworks for Intelligent CPS

Finally, modern distributed CPS are increasingly geared towards AI abstractions. New challenges emerge as articulated in recent publications on emerging intelligent CPS/IoT applications [81]. To enable language model agents to operate beyond passive reasoning and actively invoke external tools, structured communication protocols have emerged to standardize how agents discover, interpret, and execute available capabilities. Such tool-use protocols replace trial-and-error prompting with interfaces that define actions, inputs, and outputs in a consistent way, decoupling AI agents from tool implementations. Examples include the Model Context Protocol (MCP) [10], the Agent-to-Agent (A2A) protocol [127], and the Agentic Networking Protocol (ANP) [9], each targeting different layers of agent-to-tool and agent-to-agent coordination. For example, an MCP-compliant tool presents itself as a set of declarative operations with machine-readable specifications, enabling LLM agents to trigger actions with reliability approaching that of traditional APIs while retaining the flexibility of natural language for high-level planning. Most current tool-use protocols for LLM agents are request-response oriented, where an agent issues a discrete call, receives a result, and

proceeds. While these protocols increasingly support asynchronous operations and notifications, their design centers on agent-initiated tool invocations with expected responses. This creates challenges when it comes to collective AI agent interactions with physical systems where sensors produce data continuously at heterogeneous rates, multiple agents need concurrent access to the same data streams, and synchronization among different agent actions is critical for proper actuation in the physical environment. There is thus a need to rethink tool-use protocols towards supporting more holistically-coordinated interactions between the agents and their environment, while ensuring safety of such collective interactions. In addition, as with other middleware, agents must be able to discover services, control knobs, and optimization levers exported by the tools available. They may need synchronized access to global context. Finally, arbitration protocols need to ensure that concurrent agents can coordinate or negotiate when issuing conflicting decisions over shared resources.

A key challenge is how to represent the spatially distributed shared environment to the multiple agents operating within it. In edge AI deployments, relevant system state is rarely localized; instead, elements of state may be fragmented across multimodal and multi-vantage sensors, analytics services that perform additional inferences on such state, and logs that record important elements for future retrieval, likely operating at different temporal and spatial resolutions. Middleware protocols must construct a coherent working context from such distributed inputs. Supporting naive agent-initiated polling is inefficient, while ensuring full-state mirroring of all state everywhere is often an expensive overkill. Context representation must be selective. In multi-agent settings, additional complexity arises from asymmetry of information: different agents may subscribe to different signals or be scoped to different regions of the system, leading to the possibility of ingesting inconsistent or outdated beliefs unless appropriate middleware synchronization policies are enacted. As such, constructing and sharing context cannot be treated as a passive data collection problem; it requires active policies for subscription, summarization, and belief reconciliation.

Agents in a CPS/IoT application may play different functional roles, such as planner versus executor, coder versus critic, or monitor versus optimizer, where task specialization improves clarity and performance. In other settings, agents are deployed as experts, each trained or configured to handle a particular modality, subsystem, or objective, allowing the system to leverage performance gains from specialization rather than relying on a single monolithic agent. Certain control workflows additionally require collaborative sequencing [33], where perception, planning, and actuation must occur in a specific order with well-defined handoff boundaries, implying not just parallel existence of agents but structured dependency in how they coordinate. Finally, long-horizon or context-heavy tasks are often decomposed into chain-of-agent pipelines [143], where one agent summarizes, filters, or reformulates context before passing it to downstream agents to stay within the context length limits of current models. These patterns impose additional requirements on how information is shared, synchronized, and distributed among cooperating agents.

Safety assurances are another challenge. The ability to handle impending safety violations calls for appropriate detection and intervention mechanisms, the ability to defer to human oversight when appropriate, and in some cases the ability to rehearse actions in a twin or simulator before acting on the physical world. When multiple agents pursue different objectives, such as performance optimization versus fault recovery, arbitration is required to reconcile conflicting recommendations. These requirements suggest that effective agentic control in CPS/IoT applications requires more than tool invocation. The execution environment must support distributed workflow safety in the presence of multiple agents.

Furthermore, the human-in-the-loop concept integrates social sensing, treating human behavior as a physical variable that can be sensed, modeled, and influenced within the system's feedback loop.

5 Formal Methods, AI, and Cyber-Physical Systems

Formal methods are mathematical and computational techniques for the design, verification, and maintenance of systems that provide provable guarantees of desired properties, including safety, performance, security, and dependability. Formal methods have been integral to the field of cyber-physical systems since its inception as an engineering discipline. *Formal modeling*, using formalisms such as extended finite-state machines, ordinary differential equations, and timed and hybrid automata, underpin the theory and tools for model-based design of CPS. *Formal specification* languages, such as linear temporal logic, metric temporal logic, signal temporal logic, and first-order logic with various background theories are widely used. Techniques for *formal verification* including reachability analysis, model checking, temporal logic falsification, statistical model checking, and theorem proving provide automation in checking whether formal models of CPS satisfy desired specifications. Finally, *formal synthesis* provides methods for generating models and implementations (or completions) of a CPS that provably satisfy a specification.

The past decade has seen the rise of a rich and vibrant intersection between formal methods for CPS, on the one hand, and AI and machine learning (ML), on the other. In this section, we explore both facets of this intersection: the integration of AI and ML into formal methods, and the use of formal methods to provide assurance for AI-based CPS.

5.1 Integrating AI into Formal Methods for CPS

In a pair of papers published over a decade ago [109, 110], one of the authors (Seshia) made a case for the integration of AI and ML with formal methods, with two key insights. The first insight is that the *key challenges in formal methods involve synthesis problems*. These synthesis problems automate tasks that have traditionally been performed manually. For example, the main challenge in formal modeling and specification is in coming up with good models and properties. Formal verification, including highly automated methods such as model checking, require one to generate various formal proof artifacts that are crucial to proving or disproving a property. For example, proofs by induction usually require one to find a strengthening of the inductive invariant, proofs by abstraction require one to generate a suitable abstract model, proofs of program termination require one to find a ranking function, and compositional proofs require one to devise a partitioning of the system into components along with assume-guarantee contracts between them. The author advocates for making this key role of synthesis explicit, via a paradigm called "verification by reduction to synthesis." In this paradigm, formal methods problems are reformulated as synthesis problems and the resulting algorithms are solved via invocations of "synthesis oracles."

The second insight is to *use AI/ML-based methods, integrated with traditional deductive formal methods, to solve the resulting synthesis problems*. In other words, the synthesis oracles are based on learning formal artifacts from multi-modal sources of data, using a variety of learning approaches. These data sources include positive and negative examples (e.g., collected via running tests), interactions with an expert human labeling data or with a formal procedure (e.g. a verifier), as well as corpora of natural language, images, video, and other sensor data, in addition to formal specifications. The family of problems that involve the synthesis of formal artifacts from data and formal specifications is called *formal inductive synthesis* [57, 110] and the family of algorithms to solve them *oracle-guided inductive synthesis*. Even in 2015, several problems in CPS design were already demonstrated to be effectively solved using this combination of formal methods and data-driven AI [110].

In the rest of this section, we describe four different applications of this methodology of integrating AI and FM for CPS design and verification. These examples span four different AI/ML

based synthesis methods as well as different phases in the design process (specification, modeling, verification, and synthesis).

5.1.1 Formal Requirement Mining. A pre-condition to the application of formal methods is to formulate a formal specification. Verification requires a specification to check the system against. Synthesis requires a specification from which to generate the implementation. However, writing formal specifications can be the most challenging part of applying formal methods, requiring expertise in mathematical logic, formal programming languages, and the application domain. In 2012, the author encountered this firsthand while interacting with researchers and engineers in the automotive industry. However, there are usually other sources of design data, including simulation/test data and design documents, from which specifications may be extracted. Thus, the specification challenge was addressed by developing a novel approach for *requirement mining* – i.e., extracting formal requirements from available design data and interaction with engineers. One of the first successful projects was a requirement mining algorithm based on the *counterexample-guided inductive synthesis* (CEGIS) technique [58, 59], a special case of oracle-guided inductive synthesis. In CEGIS, one synthesizes a program or other formal artifact from a combination of positive examples, typically input-output traces, and counterexamples, which are traces from a verifier/falsifier showing why a candidate program is incorrect with respect to a specification. In the requirement mining approach, a specification is being reverse-engineered from a system description. More precisely, in the cited project [58], the "program" being synthesized was the specification, and it was checked against a model (in Simulink) of the automotive system under analysis. The requirements were generated in signal temporal logic (STL), from templates in parametric STL (PSTL), and vetted by an engineer (after expressing them in natural language). The learning algorithm was a custom procedure for finding the strongest instantiation of a PSTL template that satisfies a set of simulation traces. The verifier was a temporal logic falsifier for STL. This requirement mining project was a success, producing STL properties that engineers found to be useful in uncovering hard-to-find corner case bugs, and facilitating industrial technology transfer to the automotive industry.

5.1.2 Learning Environment Models. Formal verification and formal synthesis both require the specification of a model of the environment with respect to which a system is to be verified or synthesized. This model can take different forms depending on the specific application of verification or synthesis. While a human expert typically has intuition about the form of an environment model, constructing a detailed model can be a very tedious and error-prone task. An example of this challenge in the field of CPS arises in the timing analysis of embedded software. Timing analysis relies on having the "right" model of the platform on which the software task runs: detailed enough to capture timing behavior and yet abstract enough to enable scalable analysis. Traditionally, these platform models have been hand-constructed in a painstaking manner. In 2008, the GameTime approach proposed a learning-based solution for platform modeling [113, 114]. GameTime used the paradigm of game-theoretic online learning coupled with SMT-based systematic test generation to repeatedly execute carefully-generated tests on the target platform (or a cycle-accurate simulator) and learn a platform model from the resulting measurements. It combines static program analysis with machine learning from dynamic measurements. The GameTime approach was one of the first AI/ML-based approaches to learning environment models for an important application of formal methods to CPS design.

5.1.3 Generating Contracts for Compositional Verification. Compositional reasoning has long been a holy grail for formal verification. It has long been accepted in the community that the best way to scale formal methods to large systems is to employ a compositional (modular) approach: break

up the large system into smaller components, write specifications for those components, formally verify that each component satisfies its specification, and then prove that the composition of those specifications entails the desired global system-level specification. The major obstacle to realizing this approach is to come up with component-level specifications that are both sufficiently precise to entail the system-level specification and sufficiently abstract to hide unnecessary detail and enable scalable automated reasoning. As advocated for by Seshia [110], the paradigm of "verification by reduction to synthesis," enabled by AI, provides a path to realizing the vision of compositional verification. For CPS, a recent successful demonstration has been for formally verifying polyglot (multi-lingual) software written using the Lingua Franca (LF) [87] coordination language introduced earlier in this article. Chen et al. [27] present PolyVer, one of the first approaches integrating formal verifiers such as UCLID5 [105, 117], Kani [31], and CBMC [29] with synthesis oracles based on LLMs to scalably verify LF programs whose reactions are implemented in languages such as C and Rust. PolyVer uses LLMs to generate interface contracts between components (pre-/post-conditions for reactions in LF), whereas Kani and CBMC are used to verify these contracts on Rust and C functions. UCLID5 is used to perform the system-level verification of temporal safety properties on the LF program using the verified contracts. PolyVer is emblematic of an emerging approach to so-called "agentic formal verification" where AI agents, typically implemented on top of LLMs, are used to combine formal verifiers with AI models to synthesize formal artifacts (such as assume-guarantee contracts) that can scalably prove or disprove that the system satisfies a specification.

5.1.4 Learning-Based Synthesis. AI and ML have been widely used to synthesize plans and controllers for various CPS applications. However, much of this work is in the setting where specifications are given as objective functions (reward or cost functions) to be optimized. A different line of work focuses on learning plans or controllers so as to satisfy formal specifications; see, for example, the literature on *safe reinforcement learning (RL)* [61]. While this provides strong guarantees of correctness, it is most valuable if one can synthesize a plan/controller that both satisfies a formal specification and optimizes a given objective function – i.e., achieves both traditional FM objectives and AI objectives. An example of this type of result is provided in recent work by Shah et al. [119], which provides an algorithmic method to train an RL policy that achieves both objectives; i.e., it finds the optimal policy among all of those that satisfy a temporal logic specification. Such a learning-based synthesis method that jointly achieves both Boolean and quantitative objectives is a rich area for future research.

5.2 Formal Methods for AI-Based CPS: the Verified AI Agenda

Verified AI is the goal of designing AI systems that have strong, ideally provable, assurances of safety and correctness with respect to mathematically-specified requirements [116]. Seshia et al. first set out this goal in 2016 [115] and surveyed progress on this topic in their 2022 paper [116]. They organized the technical challenges for the verified AI agenda into five main topics and also covered the main principles for addressing these challenges. In the rest of this section, we briefly review of these topics in the context of AI-enabled CPS.

Environment Modeling. AI-enabled CPS operate in environments that can be extremely complex, with a great deal of uncertainty, not just about the behavior (dynamics) of agents, but also regarding which objects and agents are present in the environment. Indeed, AI is often introduced into these systems to better deal with this complexity and uncertainty. In order to perform environment modeling for these systems, Seshia et al. [116] suggest a three-pronged approach. First, they suggest using probabilistic programming languages, such as Scenic [41, 42], to capture the stochastic nature of these environments using an expressive yet interpretable formalism. Second, these formalisms are neuro-symbolic, allowing for embedding learned models of the world (e.g., of human behavior) into

the environment models. Third, they recommend using the paradigm of *introspective environment modeling* [111] to handle the uncertainty about what entities are present in the environment by algorithmically generating environment models as assumptions sufficient to guarantee safety and other key properties of the system.

Formal Specification. The task of formal specification presents unique challenges for AI systems. Many AI tasks, involving vision or natural language processing, are hard or even impossible to formalize; yet, we must reason about them to ensure the safe operation of AI-enabled CPS such as autonomous vehicles. Second, AI systems do not necessarily have the traditional, Boolean specifications that formal methods deal with; instead, the specifications tend to be more complex, often multi-objective, combining Boolean objectives with quantitative ones. Finally, properties of AI models are often just specified using data. In order to deal with all of these challenges, Seshia et al. [116] propose deriving model-level properties from system-level ones, developing hybrid Boolean-quantitative formalisms, and developing methods to learn formal specifications from data and interaction. Several classes of properties of AI/ML models have also been identified and formalized [112].

Complexity of AI Models and Systems. The complexity of AI/ML models has grown dramatically over the past decade, rising from neural models with tens of thousands of parameters to trillion-parameter models. In order to deal with this complexity, and handle the self-adaptive nature of many AI models, one needs to develop new abstractions and semantic representations of AI models. Seshia et al. [116] outline several proposals that have been successful at developing such representations that enable tractable analysis. Additionally, they advocate for developing methods to explain and interpret the behavior of AI models, for example, by distilling a formal model for mechanistic interpretability from the AI model.

Scalable Verification and Training. Effective verification and synthesis of AI systems requires novel algorithms that handle stochastic behavior of models and environments, novel specifications, and the growing complexity of AI models. To handle this, Seshia et al. [116] advocate for a compositional approach (e.g., see [138]) along with novel techniques for randomized and optimization-driven exploration of the input and state space (e.g., [35]) that uses abstract, semantic representations to scale up. This approach has been successfully applied to industrial-scale AI-enabled CPS (e.g., see [40, 43]).

Correct-by-Construction Design. In order to design an AI system to satisfy desired properties, new design paradigms are needed. For example, one needs an approach to designing systems in a data-driven manner (as in conventional training of AI systems) while satisfying a formal specification. The reader will note that this is exactly what the paradigm of *formal inductive synthesis*, identified earlier in this section, achieves. Seshia et al. [116] advocate for the use of oracle-guided inductive synthesis (oracle-guided learning) for the design of correct-by-construction AI systems; the aforementioned work by Shah et al. [119] is a good example of such an approach. Similarly, techniques for certified machine learning and safe learning-based control [128] are also highly relevant in this context. Additionally, one needs techniques for runtime assurance, such as runtime monitors, guardrails, safety filters, shields, etc., to ensure that the system remains safe even when the environment behaves in ways that are out-of-distribution or deviate from the design-time specification. See [129, 130] for more details on this topic.

6 Conclusion

Cyber-physical systems demand a shift in what we mean by a correct computation. When software mediates sensing and actuation, the right answer at the wrong time is the wrong answer; programs that must not halt, that interact continuously with their environment, and that run concurrently or

across networks cannot be judged solely by classical algorithmic criteria. The perspectives surveyed in this article address complementary facets of that challenge.

At the foundation, models of computation must elevate timing, interaction, and deterministic concurrency, and distributed coordination must make consistency and timeliness explicit rather than hoping nondeterministic middleware will suffice. Predictability then requires both disciplined design—synchronous reactive languages, logical execution time, and schedulability theory—and complementary verification, from timed automata and symbolic model checking to hybrid-system analysis. At city and IoT scale, programming and middleware abstractions must lift the designer from node-level details to mission-level sense–act loops, allocating sensing, communication, and computation against decision deadlines and data freshness. As AI enters the loop, formal methods and machine learning become mutual partners: inductive synthesis can generate the artifacts that verification and compositional reasoning need, while verified AI techniques aim to keep learning-enabled controllers inside a mathematically grounded safety envelope.

A unifying lesson is therefore that no single tool or language will suffice. We need a layered discipline: predictable cyber foundations and verified timing contracts; scalable, decision-aware connectivity; and formal assurance that scales to AI-enabled autonomy. Realizing that stack—and closing the remaining gaps in hardware support for timing, compositional verification of polyglot distributed software, and safe multi-agent actuation—is the agenda for the next generation of CPS research and practice.

References

[1] T. Abdelzaher, M. T. A. Amin, A. Bar-Noy, W. Dron, R. Govindan, R. Hobbs, S. Hu, J.-E. Kim, J. Lee, K. Marcus, et al. Decision-driven execution: A distributed resource management paradigm for the age of iot. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1825–1835. IEEE, 2017.

[2] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, et al. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 582–589. IEEE, 2004.

[3] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, second edition, 1996.

[4] R. Alur, L. de Alfaro, and T. A. Henzinger. Event-recording automata. In *Formal Methods in System Design*, pages 7–35, 1999.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–11, 1993.

[7] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A tool for modelling and implementation of embedded systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 460–464, 2002.

[8] S. Andalam, P. S. Roop, and A. Girault. Predictable multithreading of embedded applications using PRET-C. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 159–168. IEEE/ACM, 2010.

[9] ANP Open Source Community. Agent Network Protocol. https://github.com/agent-network-protocol/AgentNetworkProtocol, 2025. Accessed: 2025-08-16.

[10] Anthropic. Model Context Protocol Specification. https://modelcontextprotocol.io/specification/2025-06-18, 2025. Accessed: 2025-10-05.

[11] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, Cambridge, UK, 2009.

[12] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. In *Real-Time Systems*, volume 2, pages 301–324, 1990.

[13] S. Bateni, M. Lohstroh, H. S. Wong, H. Kim, S. Lin, C. Menard, and E. A. Lee. Risk and mitigation of nondeterminism in distributed cyber-physical systems. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2023.

[14] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Priced timed automata: Algorithms and applications. In *Formal Methods in System Design*, pages 121–151, 2005.

[15] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *Proceedings of the 8th International Conference on Computer*

Aided Verification (CAV), pages 401–413, 1996.

[16] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a tool suite for automatic verification of real-time systems. In *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 232–252, 1996.

[17] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2004.

[18] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[19] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE cloud computing*, 1(3):81–84, 2014.

[20] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[21] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[22] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv:2303.12712v1 [cs.CL]*, 2023.

[23] B. Burns, J. Beda, and K. Hightower. *Kubernetes: up and running: dive into the future of infrastructure.* O'Reilly Media, 2019.

[24] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, second edition, 2005.

[25] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, 4 edition, 2024.

[26] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR)*, pages 238–252, 2005.

[27] P. Chen, S. Lin, A. Godbole, R. Singh, E. Polgreen, E. A. Lee, and S. A. Seshia. Polyver: A compositional approach for polyglot system modeling and verification. In A. Irfan and D. Kaufmann, editors, *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design, FMCAD 2025, Menlo Park, CA, USA, October 6-10, 2025*. TU Wien Academic Press, 2025.

[28] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188, 2007.

[29] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[30] A. David, K. G. Larsen, A. Skou, and B. Nielsen. Modelling and verification of a field bus protocol using UPPAAL. In *Proceedings of the 6th International Workshop on Formal Techniques for Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 331–346, 2000.

[31] R. Delmas, Z. Hassan, Q. Hu, R. Kumar, F. R. Monteiro, T. Nguyen, A. Palacios, C. Val, M. Tautschnig, J. Adam, et al. Kani: A model checker for Rust. *arXiv preprint arXiv:2607.01504*, 2026.

[32] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(2):859–866, 1972.

[33] Z. Ding, Z. Liu, Z. Fang, K. Su, L. Zhu, and Z. Lu. Multi-Agent Coordination via Multi-Level Communication. *Advances in Neural Information Processing Systems*, 37:118513–118539, Dec. 2024.

[34] A. Donzé, O. Maler, E. Bartocci, D. Nickovic, R. Grosu, and S. Smolka. On temporal logic and signal processing. In S. Chakraborty and M. Mukund, editors, *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume LNCS 7561, pages 92–106. Spring-Verlag, 2012.

[35] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia. VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *31st International Conference on Computer Aided Verification (CAV)*, July 2019.

[36] F. Dressler, I. Dietrich, R. German, and B. Krüger. A rule-based system for programming self-organized sensor and actor networks. *Computer Networks*, 53(10):1737–1750, 2009.

[37] J. Dunkel. On complex event processing for sensor networks. In *2009 International Symposium on Autonomous Decentralized Systems*, pages 1–6. IEEE, 2009.

[38] EdgeX Foundry. https://www.edgexfoundry.org/.

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

[40] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia. Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI. In *32nd International Conference on Computer Aided Verification (CAV)*, July 2020.

[41] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: A language for scenario specification and scene generation. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2019.

[42] D. J. Fremont, E. Kim, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: A language for scenario specification and data generation. *Machine Learning*, 112(10):3805–3849, 2023.

[43] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta. Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *23rd IEEE International Conference on Intelligent Transportation Systems (ITSC)*, Sept. 2020.

[44] M. L. Gambo, A. Danasabe, B. Almadani, F. Aliyu, A. Aliyu, and E. Al-Nahari. A systematic literature review of dds middleware in robotic systems. *Robotics*, 14(5):63, 2025.

[45] N. Garg. *Apache kafka*. Packt Publishing, 2013.

[46] V. Goranko and A. Rumberg. Temporal logic. *The Stanford Encyclopedia of Philosophy*, Summer 2025 Edition, 2025.

[47] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems*, pages 126–140. Springer, 2005.

[48] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[49] K. Havelund, A. Skou, K. G. Larsen, and K. G. Nielsen. Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–11, 1997.

[50] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. Energy-efficient surveillance system using wireless sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 270–283, 2004.

[51] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The next generation. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS)*, pages 56–65, 1995.

[52] T. A. Henzinger, C. M. Kirsch, and S. A. Seshia. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[53] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *Proceedings of the IEEE*, 95(1):242–247, 2007.

[54] C. Hewitt. What is commitment? physical, organizational, and social. In *The AAMAS06 Workshop on Coordination, Organization, Institutions and Norms in agent systems (COIN)*, 2006.

[55] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[56] Eclipse ioFog. https://iofog.org/.

[57] S. Jha and S. A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *Acta Informatica*, 54(7):693–726, 2017.

[58] X. Jin, A. Donzé, J. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC)*, April 2013.

[59] X. Jin, A. Donzé, J. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 34(11):1704–1717, 2015.

[60] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.

[61] K. Jothimurugan, S. Bansal, O. Bastani, and R. Alur. Specification-guided reinforcement learning. In G. J. Pappas, P. Ravikumar, and S. A. Seshia, editors, *International Conference on Neuro-symbolic Systems (NeuS)*, volume 288 of *Proceedings of Machine Learning Research*, pages 316–330. PMLR, 2025.

[62] D. Kahneman. *Thinking Fast and Slow*. Farrar, Straus and Giroux, New York, 2011.

[63] J.-E. Kim, T. Abdelzaher, L. Sha, A. Bar-Noy, R. L. Hobbs, and W. Dron. Decision-driven scheduling. *Real-Time Systems*, 55(3):514–551, 2019.

[64] M. Kostal and M. Sojka. PRET-ization of uRISC core. In *Conference on Computer Science and Intelligence Systems (ACSIS)*, pages 495–500, 2021.

[65] T. S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, IL, 1962.

[66] A. Kurniawan. *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. Packt Publishing Ltd, 2018.

[67] S. Laddad, C. Power, M. Milano, A. Cheung, N. Crooks, and J. M. Hellerstein. Keep CALM and CRDT on. *arXiv*, 2210.12605v1 [cs.DB], 2022.

[68] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[69] K. G. Larsen, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 14–23, 1997.

[70] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[71] E. A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 363 – 369. IEEE, 2008.

[72] E. A. Lee. *The Coevolution: The Entwined Futures of Humans and Machines.* MIT Press, Cambridge, MA, 2020.

[73] E. A. Lee. What can deep neural networks teach us about embodied bounded rationality. *Frontiers in Psychology*, 25, 2022.

[74] E. A. Lee. Deep neural networks, explanations, and rationality. In B. Steffen, editor, *Bridging the Gap Between AI and Reality (AISoLA 2023)*, volume LNCS 14380. Springer, 2023.

[75] E. A. Lee. Certainty vs. intelligence. In *AISoLA: Bridging the Gap Between AI and Reality*, volume LNCS 15217. Springer, 2024.

[76] E. A. Lee. Logical time in actor systems. In J. Meseguer, C. A. Varela, and N. Venkatasubramanian, editors, *Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Prof. Gul Agha to Celebrate his Scientific Career*, volume LNCS, to appear. Springer, 2025.

[77] E. A. Lee, R. Akella, S. Bateni, S. Lin, M. Lohstroh, and C. Menard. Consistency vs. availability in distributed cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(5s):1–24, 2023.

[78] E. A. Lee, M. Lohstroh, and C. Menard. Generalizing logical execution time. In *ACM Transactions on Embedded Computing Systems*, 2022. To appear.

[79] E. A. Lee, J. Reineke, and M. Zimmer. Abstract PRET machines. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[80] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach.* MIT Press, 2011.

[81] J. Li, H. Wu, H. Zhao, T. Kimura, D. Kara, T. Wang, Y. Chen, T. Wang, Y. Hu, A. Misra, et al. Agentic llm-assisted edge ai for cps/iot applications. In *2025 IEEE 7th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 371–380. IEEE, 2025.

[82] S. Li and E. A. Lee. A preliminary model of coordination-free consistency. *arXiv*, 2504.01141 [cs.DC], 2025.

[83] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 81–95, 1998.

[84] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[85] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *International Conference on Computer Design (ICCD)*, pages 87–93. IEEE, 2012.

[86] M. Lohstroh, E. A. Lee, S. Edwards, and D. Broman. Logical time for reactive software. In *Workshop on Timing-Centric Reactive Software (TCRS), in Cyber-Physical Systems and Internet of Things Week (CPSIoT)*. ACM, 2023.

[87] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(4):Article 36, 2021.

[88] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee. Invited: Actors revisited for time-critical systems. In *Design Automation Conference (DAC)*, 2019.

[89] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(3):543–576, 2006.

[90] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[91] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 335–346, 2008.

[92] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, 2002.

[93] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, Berlin, 1992.

[94] R. Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, NJ, USA, 1989.

[95] R. Milner. Elements of interaction. *Communications of the ACM*, 36:78–89, 1993.

[96] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology, 1983.

[97] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[98] S. Natarajan and D. Broman. Temporal property-based testing of a Timed C compiler using time-flow graph semantics. In *Forum on specification and Design Languages (FDL)*, 2020.

[99] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, 2007.

[100] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, pages 78–87, 2004.

[101] A. B. Noel, A. Abdaoui, T. Elfouly, M. H. Ahmed, A. Badawy, and M. S. Shehata. Structural health monitoring using wireless sensor networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 19(3):1403–1423, 2017.

[102] L. M. Oliveira and J. J. Rodrigues. Wireless sensor networks: A survey on environmental monitoring. *J. Commun.*, 6(2):143–151, 2011.

[103] Open Yurt. https://openyurt.io/.

[104] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977. According to Alur and Henzinger, the first application of temporal logic (tense logic) to reactive systems.

[105] E. Polgreen, K. Cheang, P. Gaddamadugu, A. Godbole, K. Laeufer, S. Lin, Y. A. Manerkar, F. Mora, and S. A. Seshia. UCLID5: multi-modal formal modeling, verification, and synthesis. In *34th International Conference on Computer Aided Verification (CAV)*, volume 13371 of *Lecture Notes in Computer Science*, pages 538–551. Springer, 2022.

[106] B. B. Rad, H. J. Bhatti, and M. Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[107] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for internet of things: a survey. *IEEE Internet of things journal*, 3(1):70–95, 2015.

[108] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009(Article ID 758480):17 pages, 2009.

[109] S. A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the Design Automation Conference (DAC)*, pages 356–365, June 2012.

[110] S. A. Seshia. Combining induction, deduction, and structure for verification and synthesis. *Proceedings of the IEEE*, 103(11):2036–2051, 2015.

[111] S. A. Seshia. Introspective environment modeling. In *19th International Conference on Runtime Verification (RV)*, pages 15–26, October 2019.

[112] S. A. Seshia, A. Desai, T. Dreossi, D. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue. Formal specification for deep neural networks. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 20–34, October 2018.

[113] S. A. Seshia and A. Rakhlin. Game-theoretic timing analysis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 575–582. IEEE Press, November 2008.

[114] S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):55:1–55:27, 2012.

[115] S. A. Seshia, D. Sadigh, and S. S. Sastry. Towards verified artificial intelligence. *CoRR*, abs/1606.08514, 2016.

[116] S. A. Seshia, D. Sadigh, and S. S. Sastry. Toward verified artificial intelligence. *Communications of the ACM*, 65(7):46–55, 2022.

[117] S. A. Seshia and P. Subramanyan. Uclid5: Integrating modeling, verification, synthesis, and learning. In *Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, October 2018.

[118] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2):101–155, 2004.

[119] A. Shah, C. Voloshin, C. Yang, A. Verma, S. Chaudhuri, and S. A. Seshia. Ltl-constrained policy optimization with cycle experience replay. *Transactions on Machine Learning Research (TMLR)*, 2025, 2025.

[120] P. Shojaee, I. Mirzadeh, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv*, 2506.06941 [cs.AI], 2025.

[121] H. A. Simon. Theories of bounded rationality. In C. B. McGuire and R. Radner, editors, *Decision and Organization*, pages 161–176. North-Holland Publishing Company, Amsterdam, 1972.

[122] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, 2006.

[123] D. Soni and A. Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, 2017.

[124] M. Stigge and W. Yi. Graph-based models for real-time workload: a survey. *Real-Time Systems*, 51(5):602–636, 2015.

[125] R. Sugihara and R. K. Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):1–29, 2008.

[126] K. Terfloth, G. Wittenburg, and J. Schiller. Facts-a rule-based middleware architecture for wireless sensor networks. In *2006 1st International Conference on Communication Systems Software & Middleware*, pages 1–8. IEEE, 2006.

[127] The Linux Foundation, A2A Project. Agent2Agent (A2A) Protocol Official Specification. https://a2a-protocol.org/v0.3.0/specification/, 2025. Accessed: 2025-08-16.

[128] C. Tomlin. Safe learning in robotics. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3–3, 2021.

[129] H. Torfah, S. Junges, D. J. Fremont, and S. A. Seshia. Formal analysis of AI-based autonomy: From modeling to runtime assurance. In *21st International Conference on Runtime Verification (RV)*, volume 12974 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2021.

[130] H. Torfah, C. Xie, S. Junges, M. Vazquez-Chanlatte, and S. A. Seshia. Learning monitorable operational design domains for assured autonomy. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*, October 2022.

[131] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997.

[132] P. Wegner, F. Arbab, D. Goldin, P. McBurney, M. Luck, and D. Roberson. The role of agent interaction in models of computing: Panelist reviews. *Electronic Notes in Theoretical Computer Science*, 141:181–198, 2005.

[133] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv*, 2201.11903 [cs.CL], 2022.

[134] M. Welsh, G. Mainland, et al. Programming sensor networks using abstract regions. In *NSDI*, volume 4, 2004.

[135] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, 2004.

[136] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer Aided Design*, 28(7):966 – 978, 2009.

[137] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium On Edge Computing (SEC)*, pages 373–377. IEEE, 2018.

[138] B. Yalcinkaya, H. Torfah, D. J. Fremont, and S. A. Seshia. Compositional simulation-based analysis of AI-based autonomous systems for Markovian specifications. In P. Katsaros and L. Nenzi, editors, *23rd International Conference on Runtime Verification (RV)*, volume 14245 of *Lecture Notes in Computer Science*, pages 191–212. Springer, 2023.

[139] W. Yi. CCS + time = an interleaving model for real time systems. In *ICALP 91*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228, 1991.

[140] W. Yi, C. Huang, B. Khodabandeloo, and D. A. Nguyen. Mimos-tools: An integrated toolchain for model-based design of embedded systems. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, Munich, Germany, Oct. 2026. ACM.

[141] W. Yi, M. Mohaqeqi, and S. Graf. MIMOS: A deterministic model for the design and update of real-time systems. In *Proceedings of COORDINATION 2022*, 2022. Also available on arXiv, 2020: https://arxiv.org/abs/2011.13234.

[142] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings of the 7th International Conference on Formal Description Techniques (FORTE)*, pages 3–18, 1994.

[143] Y. Zhang, R. Sun, Y. Chen, T. Pfister, R. Zhang, and S. Ö. Arık. Chain of Agents: Large Language Models Collaborating on Long-Context Tasks. *Advances in Neural Information Processing Systems*, 37:132208–132237, Dec. 2024.

[144] 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.