INVITED PAPER

A Platform Programming Paradigm for Heterogeneous Systems Integration

By Kai-Björn Gemlau, Leonie Köhler, and Rolf Ernst, *Fellow IEEE*

ABSTRACT | To cope with growing computing performance requirements, cyber–physical systems architectures are moving toward heterogeneous high-performance computer architectures and networks. Such architectures, however, incur intricate side effects that challenge traditional software design and integration. The programming paradigm can take a key role in mastering software design, as experience in automotive design demonstrates. To cope with the integration challenge, this industry has started introducing a programming paradigm that efficiently preserves application data flow under platform integration and changes with minimum performance loss. This article will revisit this paradigm that is currently used for lock-free multicore programming and explain its extension to the system level. It will then explore its application to two important developments in industrial design. This article will conclude with an evaluation of its properties, its overhead, and its application toward a robust design process.

KEYWORDS | Automation; industry 4.0; real-time (RT) programming abstractions; smart manufacturing; system integration; system-level logical execution time (SL LET).

NOMENCLATURE

ADAS	Advanced driver assistance system.
API	Application programming interface.
BCET	Best case execution time.
BCRT	Best case response time.
BET	Bounded execution time.
BSW	Basic software.
CDR	Common data representation.
CPS	Cyber–physical system.
DAG	Directed acyclic graph.
DCPS	Data-centric publisher-subscriber.
DDS	Data distribution service.
DDSI	DDS interoperability wire protocol.
ECU	Electronic control unit.
EDP	Endpoint discovery protocol.
ESP	Electronic stability program.
FB	Functional block.
FRER	Frame replication and elimination for reliability.
GALS	Globally asynchronous locally synchronous.
GPGPU	General purpose computation on graphics processing unit.
HMI	Human–machine interface.
HPC	High-performance computing.
HRI	Human–robot interaction.
HTL	Hierarchical timing language.
IIoT	Industrial Internet of Things.
I/O	Input/output.
IP	Internet protocol.
IPMCS	Industrial process measurement and control system.
IRQ	Interrupt request.
IRT	Isochronous real time.
LET	Logical execution time.
LTTA	Loosely time-triggered architecture.
MDD	Model-driven development.
MPSoC	Multiprocessor systems-on-chip.
NTP	Network time protocol.
NUMA	Nonuniform memory access.
OEM	Original equipment manufacturer.
OPC UA	OPC unified architecture.
OSI	Open systems interconnection.
OS	Operating system.

Manuscript received March 21, 2020; revised August 21, 2020; accepted October 11, 2020. This work was supported in part by the German Research Foundation (DFG) under Grant ER168/30-2, in part by the German Federal Ministry of Education and Research (BMBF) under Grant 16EM00285, and in part by the Research Contract from Daimler. *(Corresponding author: Kai-Björn Gemlau.)*

The authors are with the Institute of Computer and Network Engineering, Technische Universität Braunschweig (TU Braunschweig), 38106 Braunschweig, Germany (e-mail: gemlau@ida.ing.tu-bs.de; koehler@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de).

Digital Object Identifier 10.1109/JPROC.2020.3035874

PDP	Participant discovery protocol.
pHRI	Physical human–robot interaction.
PIM	Platform-independent model.
PLC	Programmable logic controller.
PSM	Platform-specific model.
PTP	Precision time protocol.
QoS	Quality-of-service.
RAMI 4.0	Reference architecture model industry 4.0.
RE	Runnable entity.
ROS 2	Robot operating system 2.
RTE	Runtime environment.
RTPS	Real-time publish subscribe protocol.
RT	Real time.
SAE	Society of Automotive Engineers.
SDN	Software defined networking.
SL LET	System-level logical execution time.
(SL) LET	(System-level) logical execution time.
SOME/IP	Scalable service-oriented middleware over IP.
SPNP	Static priority nonpreemptive.
SPP	Static priority preemptive.
SW-C	Software component.
SysML	Systems modeling language.
TAI	Temps Atomique International.
TAS	Time aware shaper.
TDL	Timing definition language.
TDMA	Time division multiple access.
TSN	Time-sensitive networking.
TTA	Time-triggered architecture.
UDP	User datagram protocol.
UML	Unified modeling language.
VFB	Virtual functional bus.
WCET	Worst case execution time.
WCRT	Worst case response time.
XML	Extensible markup language.
ZET	Zero execution time.

I. INTRODUCTION

CPSs are distributed hardware–software systems which control and monitor physical processes [1]. All systems industries rely on CPS technology including industrial automation, automotive, or aviation technology, to name just a few. To cope with growing computing performance requirements, CPS architectures increasingly resort to high-performance computer architectures, such as MPSoCs, GPGPUs, or accelerators for machine learning. Together with their related memory hierarchies, they are combined with heterogeneous hardware architectures.

Programming such hardware platforms is coming closer to parallel programming in HPC, but there are major differences. CPSs consist of large, interdependent application functions and services that are subject to timing and often safety requirements. Usually, many such applications share the same platform, requiring separation of function and timing. Software technology is struggling to keep up with this growing hardware and application complexity, just like communication technology that must transport and synchronize huge amounts of data.

A consequence of heterogeneity is the coexistence of different parallel programming styles for different components. Then, component integration requires software integration that leads to predictable behavior at the system level. Such software integration should be robust under function changes and portable to updated architectures. It should be controllable by the system programmer.

A suitable basis for such integration is found in the LET programming paradigm [2]. Starting from a concept for timed programming, LET has been developed into a lock-free programming and integration paradigm for shared memory multicore architectures in automotive applications [3], [4]. It has been standardized and successfully introduced in series development at many automotive companies. However, LET does not scale beyond tightly integrated components.

In this article, we explain how to derive a system-level extension of LET that has the needed properties of a system-level CPS programming paradigm. We use a robotics example to motivate its use in smart manufacturing. It addresses key problems of data synchronization and versioning in distributed systems enabling subsystem composition with predictable and robust timing.

This article is organized as follows. To begin with, we discuss a case study from the application area of smart manufacturing [5] and highlight the functional properties that have to be preserved during integration. Since the issues in the design process are not limited to the industrial domain, we give an overview of related work and review how the automotive domain has dealt with them so far. This includes, in particular, the SL LET programming, which is introduced in Section IV together with a summary of its benefits in the development process of distributed CPSs. We further show in Section V how the SL LET concept fits state-of-the-art middleware enabling deterministic timing and data flow among participants. In addition, we explain in Section VI how SL LET can be used to handle the complexity in the design space of TSN, which is a common communication backend for modern CPSs.

II. PROBLEM STATEMENT

In this section, the need for a platform programming paradigm for heterogeneous systems integration is motivated and illustrated by a pHRI use case.

The pHRI example in Fig. 1 describes a scenario in which a robot holds a potentially heavy workpiece while a human is working on it [5]. Consequently, it is not acceptable to simply stop the robot as soon as the contact between them is detected. Instead, the trajectory of the robot, the position of the objects in the workspace, and the human–machine interactions have to be closely monitored and controlled.

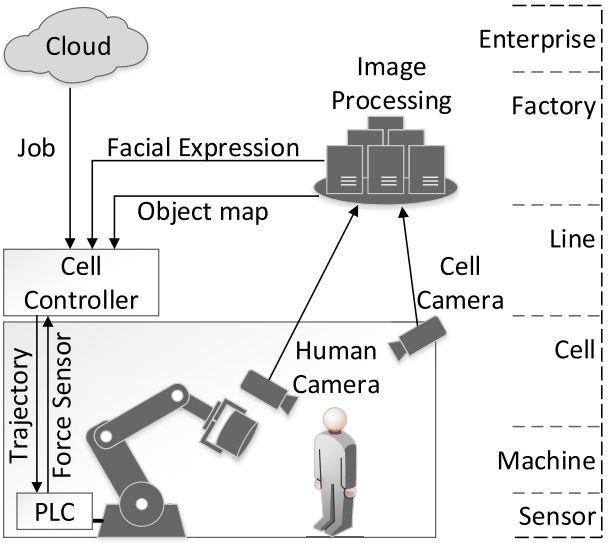


Fig. 1. ***Different hierarchy levels in the pHRI use case.***

A typical realization would include different levels of factory automation: due to their complexity, jobs such as image processing are suitable candidates for centralized computing (also called cloud or fog computing) [6]. An example would be a central image-processing hardware per production line or even beyond. All camera streams are sent to a high-performance computer that produces a 3-D map of objects as well as the detected facial expression of the worker. At the same time, the specific production job (e.g., for individualized products) comes from a cloud application. The cell controller has to merge this information with data from the robot sensors to plan the robot trajectory.

The development of a CPS starts typically with a high-level, often executable description of the system functionality. A popular modeling language for the functional simulation of CPS applications is defined in the industrial standard IEC 61499 [7]. Applications are here described in terms of connected FBs with both event and data inputs/outputs. Each FB represents a function that reads inputs and produces outputs. There are two relevant mechanisms which may trigger the execution of an FB: It can be activated periodically (time triggering), or it can be activated by an input event (event-based triggering). A chain of linked FBs is called cause–effect chain and describes a sequence of processing steps.

Fig. 2 provides an exemplary application model with three FBs in the notation of IEC 61499. They are activated periodically with a multiple of a base period and have register-like communication, which means that the output of an FB is overwritten by the consecutive execution of the FB. The two involved cause–effect chains are marked in red and green.

Often requirements relate to cause–effect chains, for example, the specification of a maximum end-to-end latency or the specification of the data flow between FBs.

A possible safety goal is that "the system must detect the human worker with its camera and adapt the robot trajectory such that the worker does not get accidentally harmed." Taking the robot velocity into account, this can be translated into a latency requirement like "it must not take longer than x milliseconds from capturing the camera frame to the reaction of the robot."

Once the functional model of the CPS is approved, it needs to be gradually refined and finally an implementation must be generated. An implementation is correct if it preserves the properties of the functional model. The choice of the RT programming paradigm mainly determines whether it is easy or difficult to translate the functional model into a correct implementation. It is desirable that an RT programming paradigm for CPSs features [8]:

1) time predictability;
2) data flow determinism;
3) composability;
4) platform independence.

Time predictability ensures that the timing properties such as latencies and jitter of an implemented cause–effect chain can be efficiently computed in advance; this serves to check the correspondence between specification and implementation.

Data-flow determinism is a property of the RT programming paradigm that helps to preserve the deterministic data flow specified in the functional model throughout the implementation.

Since the functional model is a composition of components which may execute in parallel, the RT programming paradigm should support concurrence and synchronization. In particular, it is desirable that component properties are preserved through composition (so-called composability) which eases not only design and verification but also the reuse of components in different system configurations.

Platform independence means that time predictability, data flow determinism, and composability are preserved through a platform modification, even if the platform is distributed and applies asynchronous communication networks. Platform independence is of importance since modern CPSs are subject to hardware and software updates as well as reconfiguration.

The four requirements prioritize a manageable design process with repeatable results over maximum performance. Take data flow determinism as an example. Using minimum age data values at all times could help to maximize automatic control performance, but data flow would change with computation timing that, in turn, depends on the execution platform, its load, temperature control, and so on. The resulting nondeterministic data flow depends on a large, time-variant parameter space that impacts system behavior. This impact challenges a systematic design with

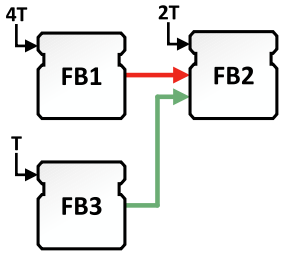


Fig. 2. ***Two cause–effect chains with oversampling and undersampling that are joined at FB2.***

predictable, platform-independent results. There might be cases where this is acceptable, but it can hardly be the basis of a manageable complex system design.

Unfortunately, the most common RT programming paradigm—namely the BET paradigm [2], which models the variability of execution times observed on conventional platforms, has limited time predictability and does not feature data flow determinism, composability, and platform independence [8]. For this reason, the resulting implementation has to be extensively tested to assure that the implemented CPS fulfills the safety requirements. A particular challenge is to preserve the data flow relations in cause–effect chains under BET. The described time-triggered communication between FBs in Fig. 2, for instance, leads to race conditions because the data to be sampled is produced with run-time varying execution speed, a problem which is aggravated with the dynamic behavior of modern processing hardware. This unsatisfactory situation has already been observed in domains other than industrial automation. Particularly, in the case of automated driving, where no human driver is able to react to failures, stringent safety requirements have to be met while function complexity grows asking for heterogeneous high-performance platforms. At the same time, automotive software development becomes more agile and new functionality, as well as updates, have to be rolled out in the field. Therefore, extensive testing has to be avoided, and *ad hoc* implementations do not scale anymore. The automotive industry has therefore introduced LET as an RT programming paradigm satisfying all of the above-named properties; it is now included in the AUTOSAR standard [9]. However, LET is applicable only in the scope of single-core and multicore processors, an extension for distributed systems was recently proposed under the name of SL LET [8] but only systems with register-like communication semantics between FBs have been discussed.

In this article, we explore whether SL LET can be applied to program systems which feature complex data flow including buffering and RT constraints. This problem is particularly relevant for building correct distributed heterogeneous CPS which includes RT middleware for communication between remote applications. The middleware and the underlying network communication are often challenging to program because timely and deterministic transport and delivery of data samples over synchronous and asynchronous networks is required. The efficiency of the presented solution with SL LET shall therefore be demonstrated by applying it to the DDS, a middleware for RT data distribution, and TSN, a set of standards for timely communication over Ethernet. In the end, the solution is evaluated including a discussion of the incurred implementation overhead.

III. RELATED WORK ON RT PROGRAMMING PARADIGMS

If a computing system has to interact with its physical environment, then the system (re)actions must be appropriately timed to achieve the desired effects. Numerous examples for such CPSs, which connect the world of information technology and the physical world, can be found both in the industrial and automotive domains. Software design for CPSs belongs to the field of RT programming because the correctness of programs depends on both the computed values and the required execution times. To describe the behavior of a program in time, different programming paradigms are in use.

Synchronous-reactive programming relies on the ZET paradigm, while the BET paradigm is widely used in the field of classic scheduling theory [10]. The LET paradigm [10], [11], which connects these two worlds, is becoming increasingly popular in the domain of automotive RT software [8]. A relatively new approach is the reactor-based programming paradigm proposed in [12].

The ZET, BET, and LET programming paradigms can be related to a basic notion of an RT process or a task model. A task is any piece of software which consumes service of the hardware element on which it runs. We assume here that a task is activated repeatedly with a period T and an offset φ, and each instance is called a job. Furthermore, let every job read inputs at the start of its execution and write outputs at its termination. A set of tasks with data flow dependences is called application in this article.

A. Bounded Execution Time

The BET paradigm assumes that the delay between the start and termination of a task τ_i, that is, its response time R_i, varies with each job due to variations in its own execution demand and variations in the interfering workload of other tasks which are scheduled on the same processing element. The BET paradigm specifies a relative deadline d_i—also called BET—for each task τ_i. A task is said to be correct if the WCRT $\overline{R}_i$ is not larger than this BET d_i, which can be verified with techniques from classic scheduling theory.

The BET paradigm is currently the prevalent paradigm for RT programming because it truthfully models the observable RT behavior of software on a conventional execution platform. Indeed there is a close correspondence between a BET task and an OS task. However,

the BET behavior of tasks satisfies only the property of time predictability: by applying the techniques from classic scheduling theory, it is possible to compute lower and upper bounds on the response times and jitter of BET tasks. The variations in the I/O timing of tasks lead to nondeterministic data flow among jobs which read and write their inputs and outputs. Moreover, the response times, the input–output timing of tasks, and thus also the data flow is not preserved in the case of composition or platform modifications.

B. Zero Execution Time

The ZET paradigm assumes that with the occurrence of an activation event, a task reads inputs, computes, and broadcasts outputs in zero time. This synchrony hypothesis is satisfied by any implementation, which can guarantee that during the execution of one job, no other job of the same or other tasks may arrive [13]. Temporal properties are implicitly expressed in terms of event occurrences. This ZET or also called synchronous-reactive behavior leads to time predictability, data flow determinism, composability, and platform independence [2]. Synchronous languages to specify an application are, for example, the imperative language Esterel [14], and the declarative data flow languages Luster [15] and Signal [16].

A synchronous application can be compiled to a finite automaton, but the compilation is difficult and the entailed program transformations impede traceability of safety requirements [17]. Given this automaton and the characteristics of the executing processor, it can be checked whether the synchrony hypothesis is valid [18].

Another issue of the synchronous reactive programming is the synthesis of distributed architectures [19], [20], in particular, if synchronous components are communicating not over a synchronous but an asynchronous network (GALS systems). For this challenging problem of desynchronization, multiclocked synchronous model of computations has been proposed [21], [22] as well as an LTTA [23]–[25].

C. Logical Execution Time

An LET task λ_i reads its inputs at the activation instant in zero time and writes its outputs in zero time at the elapse of a constant logical execution time LET_i. In comparison to the ZET paradigm, the synchrony assumption is restricted to the zero-time reading and writing of inputs and outputs. The finite execution time eliminates the zero-delay loops which lead to compilation problems in synchronous reactive programming. Different languages formally describe the semantics of LET tasks, where Giotto [26] was the first, while HDL [27] and TDL [28] extend the expressive capabilities by adding aspects like hierarchical LET applications.

The LET paradigm enforces a deterministic I/O-timing of tasks which leads to predictable response times, zero jitter, deterministic data flow, as well as composability and platform independence [2].

Obviously, the LET programming paradigm does not correspond to the behavior of an OS task on a conventional execution platform. However, it can be implemented with little effort by an additional, lightweight software layer as has been demonstrated in different works [3], [4], [29] even for relevant platforms in the automotive industry. The principle of those implementations is that: 1) task outputs are buffered until they are published with the elapse of the LET by some driver and 2) the inputs of a task are copied immediately at its activation by some driver. If several driver actions coincide, then first all write actions, and then all read actions are performed to maintain causality. The synchrony assumption, that driver actions occur in logically zero time, is satisfied by an implementation, if during driver execution no new events may occur [29].

The LET paradigm pays for its desirable properties. Response time variations cannot be exploited since the LET of a task must be chosen equal to or larger than its WCRT. Moreover, buffers are required which store outputs until they are to be published [30]. This has prevented designers from applying the LET to the moment when architectural complexity asked for a programming paradigm that leads to deterministic timing and data flow and enables MDD while being compatible with legacy software and hardware artifacts. The LET paradigm has been a success in the automotive industry because it elegantly solves the problem of multiprocessor programming with lock-free inter-core communication [31]. It should be noted that LET by itself does not increase the workload, even though early implementations might have used a nonworkload-preserving scheduling strategy (see Section VII). Biondi and Di Natale [3] also pointed out that due to the predictable timing of LET, memory accesses can be optimized and contention avoided. The LET paradigm is standardized in the AUTOSAR timing extensions [9].

D. Reactors

Lohstroh *et al.* [12] and Lohstroh and Lee [32] very recently proposed the reactor-based programming paradigm which is a deterministic model for distributed systems. Reactors are concurrently executing objects that exchange timestamped messages. Reactors are built from reactions, which are triggered on message reception and may produce an output message. Timestamping, which is performed in terms of logical time, serves to deterministically order the consumption of messages by reactions. The proposed paradigm thus achieves data flow determinism, and the data flow is also composable and platform-independent. In contrast to (SL-) LET, the concept does not imply time determinism. (SL-) LET uses time determinism as an implicit way of timestamping, which has been proved to be both intuitive and practical. It also enables efficient implementations as shown in several studies in

the context of the automotive industry, and we will also see further examples in Sections IV–VI in this article. In principle, reactors should be time-predictable, but currently no scheduling analysis has been demonstrated. It has only been mentioned that run-time errors violating the deadline can be detected. The integration in a design flow and the integration of legacy components also remain unclear given the novelty of the concept.

IV. SYSTEM-LEVEL LET: AN RT PROGRAMMING PARADIGM FOR DISTRIBUTED SYSTEMS

As motivated in the problem statement, we want to investigate in this article whether an LET-like programming paradigm, which was already applied successfully in the context of automotive systems, can also be a useful tool for distributed industrial CPSs. The challenge here is to support the implementation of RT middleware required for communication between remote applications in industrial CPSs. Such a middleware typically features complex data flow with buffering and RT constraints.

The LET programming paradigm itself is not immediately applicable to distributed systems since the assumed zero-time I/O hypothesis conflicts with the nonnegligible delays of communication between remote nodes. A possibility is to include the communication delay within the LET of a task as proposed in [33]–[35]. This leads to transparent distribution in the sense of [33], however, it may restrict the design space: not only that the transmission delay can never be larger than the LET of the sender task, but also the frequency of message transmission is forced to equal the period of the sender task. Moreover, this approach mixes application design and network design which are separate fields both from a technical and an organizational perspective. SL LET [8] extends the LET programming paradigm to distributed systems by introducing the notion of time zones and explicit LET interconnect tasks.

Concept, implementation, and benefits of SL LET are presented in this section, while Sections V and VI discuss how to program RT middleware and the underlying network communication over TSN with SL LET.

A. Concepts

The SL LET programming paradigm relates to distributed RT systems with a bounded synchronization error among the clocks of its nodes. An LET application, that is to be executed on the hardware platform of this RT system, consists of a set of communicating LET tasks.

Once the basic architecture of the hardware platform and the mapping of the software functions to this platform is decided, communication can be categorized as local and remote. Local communication is characterized by the fact that it can be performed in logically zero time, whereas remote communication has too long delays for the synchrony hypothesis to be fulfilled.

SL LET decomposes an LET application into distinct subsets of LET tasks, which are called time zones, such that for all LET tasks within a time zone local communication applies and the classic LET programming paradigm is valid.

Definition 1 (Time Zone [8]): A time zone decomposition of an LET application is a partition of the set of LET tasks. Each element of the partition is called time zone Z.

All LET tasks in the same time zone share a local clock, which is approximate of the global clock. The maximum time difference between any two local clocks is bounded from above by a known finite error ϵ.

Definition 2 (Synchronization Error [8]): The maximum error between a time instant $t_i^{Z_a}$ in time zone Z_a and the same time instant $t_i^{Z_b}$ in time zone Z_b is bounded by

$$\forall a, b: \; t_i^{Z_b} - \epsilon \leq t_i^{Z_a} \leq t_i^{Z_b} + \epsilon. \tag{1}$$

The remote communication is realized by a so-called SL LET interconnect task, which is a conventional LET task copying data between a pair of remote time zones.

Definition 3 (Time Zone Interconnect [8]): An SL LET interconnect task Φ is an LET task that copies an output, which is stored in a memory m in a time zone Z_a to a memory m' in a remote time zone Z_b.

Note that SL LET does not constrain the low-level implementation of remote communication but makes the I/O interface time-deterministic.

The broadcasting of task outputs in logically zero time thus only applies within a time zone, whereas the SL LET interconnect task is charged to deterministically deliver data from remote nodes and deal with the problem clock synchronization.

B. Implementation

The SL LET interconnect task is the main element which distinguishes SL LET-based implementation of a distributed RT system from the implementation of the same system with LET programming on the individual nodes and BET-like communication between them.

An SL LET interconnect task is, like an LET task, an abstract concept which can but does not have to map to a single OS task. In the common case, the SL LET interconnect task describes the end-to-end timing of a remote communication including the communication stacks in the sender time zone and the receiver sender time zone as well as the transmission on the network. Since the communication stacks and the network usually do not behave in a time-deterministic manner, the desired behavior must be enforced.

First of all, SL LET requires that all local LET schedules are synchronized with regard to a global clock, which can be realized by applying the PTP defined in IEEE 1588v2 [36] and starting the schedules with the occurrence of IRQs [4]. Time synchronization among remote

nodes is common in distributed systems and can thus not be considered as an implementation overhead that is specific to SL LET.

Second, to hide varying network transmission latencies and out-of-order arrivals of packets, a possible strategy is to introduce sequence numbering as well as buffering and reordering in the receiving time zone [8]. Although the addition of a sequence number in a packet is an insignificant overhead, the overhead related to buffering is of interest. Gemlau *et al.* [8] showed that the number of required buffer entries N in the receiving time zone which receives data from an SL LET interconnect task with period T depends on the maximum network jitter WCRT $-$ BCRT, the maximum time required to read an entry from a buffer Δ_R, and the relative clock synchronization error ϵ between the communicating time zones. If LET_Φ is chosen at least as large as the maximum transmission time plus the synchronization error such that $\mathrm{LET}_\Phi \geq \mathrm{WCRT} + \epsilon$, we have [8]

$$N = 1 + \left\lceil \frac{\mathrm{LET}_\Phi - \mathrm{BCRT} + \Delta_R + \epsilon}{T} \right\rceil . \tag{2}$$

The relation shows not only that the memory overhead is constant, but also that

1) the accuracy of the synchronization error and
2) the jitter $\mathrm{LET}_\Phi - \mathrm{BCRT}$

can be traded for the number of buffer entries and vice versa.

C. Benefits of SL LET in the Design Flow and Verification Process

SL LET is not only an RT programming paradigm but supports in many respects an efficient design flow and verification process which we will illustrate in this section and later in Sections IV–VI by the pHRI use case introduced in Section II which has stringent safety and timing requirements.

1) SL LET Supports MDD: With the increasing complexity of modern distributed CPSs both with regard to the software architecture and the execution platform, an *ad hoc* approach to the design of such systems with high-performance requirements does not scale anymore. An example of the increased complexity is the shift from federated to integrated architectures [37] which is accompanied by the transition from single-core to multicore controllers and the continued replacement of data buses by networks. In the future, these networks will also be used to connect the upper hierarchical levels of factory automation to the control level and below [38]. At the same time, it is desirable to reuse functionality and to be able to modify platform configurations. Therefore, the design and implementation process should feature compositionality and platform independence.

MDD is a way of structuring and ultimately automating the design process [39]. The development starts with models of the system functions at a high level of abstraction to unburden the application design from implementation details. Such functional models are typically written in domain-specific languages with support for graphical entry and are often executable to enable functional simulation. Generic examples for such languages are MATLAB/Simulink, LabView, and others, but also the important industrial standard IEC 61499 [7] proposes a modeling language for applications of CPSs in terms of connected FBs with both event and data inputs/outputs. The quoted functional languages have in common that they have synchronous-reactive semantics or can be associated with such. In the important case of the standard IEC 61499, which was published with ambiguous semantics [7], compatible synchronous semantics for the execution of FBs have been proposed [40].

The top layer of Fig. 3 shows a coarse-grained functional model of the pHRI. According to IEC 61499, it is built from FBs (FB1–FB9), including data flow (bold lines) and event flow (arrows).
Individualized products are a typical industry 4.0 scenario, therefore FB1 represents the current manufacturing job that may change for each workpiece. The production cell is equipped with a set of sensors like stereo or RGB-D cameras and laser scanners (FB2) that feed the object detection (FB5), to get a model of every object that is inside the production cell. On the other hand, the human worker can also be monitored, for example, by a camera for facial expression or head position or by body sensors for heart rate (FB3, FB6). This provides information about the workers' intentions and their potential fatigue. The trajectory planning FB7 combines these results with sensor data from the robot (like force, ultrasonic distance sensors, or actuator fault sensors) to compute a safe but efficient trajectory for the robot arm. The following collision detection (FB8) acts as a safety fallback that has to decide between a potential hazardous collision or an intended contact.
The execution of an FB might be either triggered periodically (e.g., the trajectory planning in FB7) or event-based (e.g., the object detection in FB5, that starts as soon as a new image from the camera arrives).

In an MDD, these functional models are then, manually or automatically, synthesized in a series of model transformations where a refined model should preserve the properties of the more abstract model and the relation between the models should be traceable.

If the SL LET programming paradigm is applied in a system, then the model transformation is significantly eased because the synchronous-reactive semantics of many functional models are compatible with SL LET given that the FBs are periodically executed. Also event-triggered

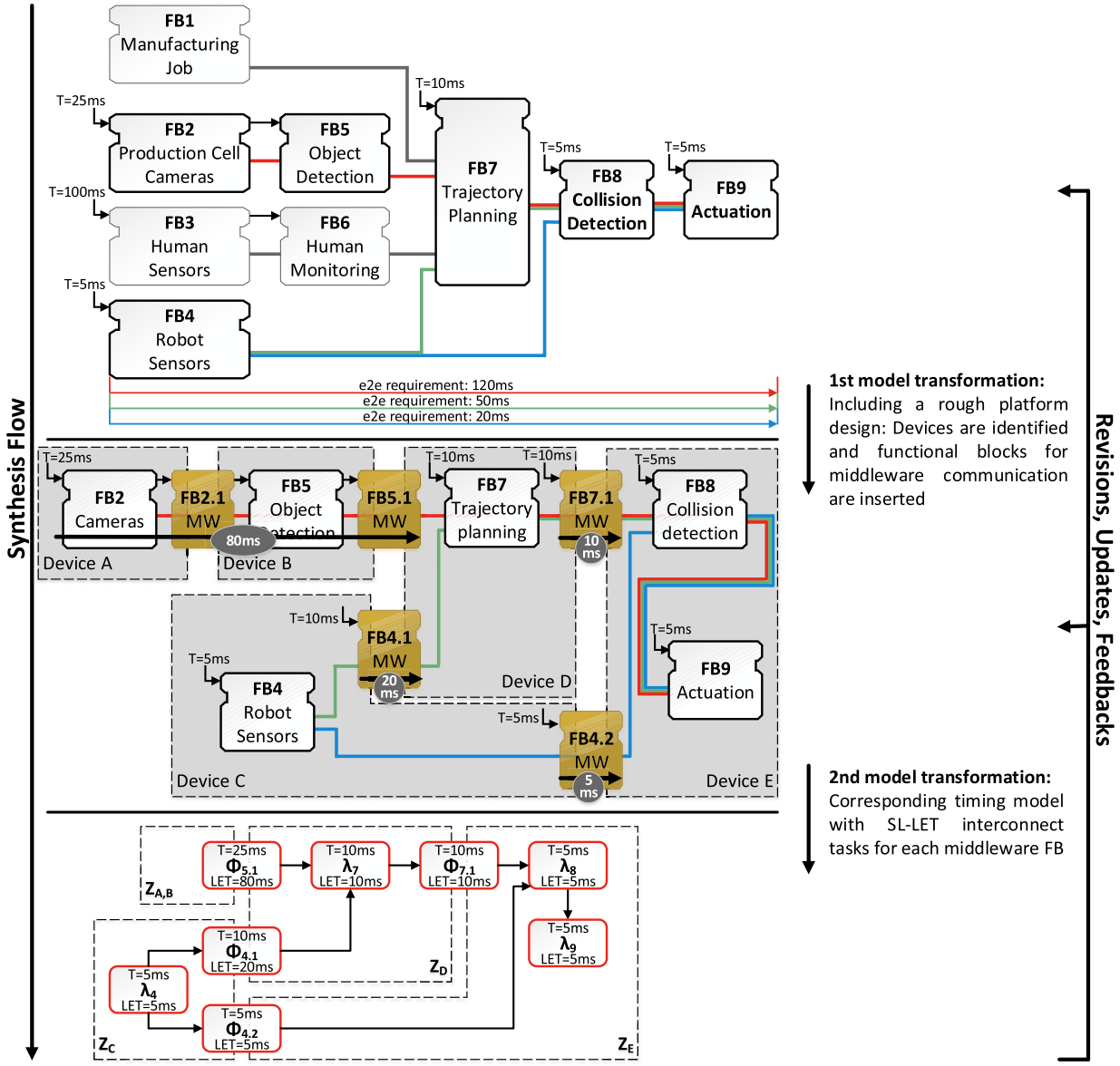


Fig. 3. ***Model-driven design process for the pHRI use case showing a functional model, a refined functional model, and a translation into the SL LET programming paradigm which can be implemented, for example, as proposed in [8].***

cause–effect chains of FBs can be treated if the first FB in such a chain is periodically activated.

As we will see in the following, SL LET allows us to address a major issue with MDD, which is to trace back design decisions to the original safety requirements and vice versa [17].

Fig. 3 shows multiple transformations and refinements in the synthesis flow of the case study. The functional model in the upper part of the figure represents the initial stage of the design process. Based on the functional model, a first simulation can be done without incorporating a specific hardware platform. This already gives a first estimation of the end-to-end latency for each cause–effect chain. In the following, we will focus on the three colored cause–effect chains, namely

1) red (FB2 $\prec$ FB5 $\prec$ FB7 $\prec$ FB8 $\prec$ FB9),
2) green (FB4 $\prec$ FB7 $\prec$ FB8 $\prec$ FB9),
3) blue (FB4 $\prec$ FB8 $\prec$ FB9),

which are supposed to have end-to-end timing requirement. These cause–effect chains of interest join and fork: the red and green one join at the trajectory planning in FB7, which combines inputs from FB5 and FB4. The green and blue one fork at the output of FB4, such that this output can be used by both FB7 and FB8.

First transformation: In the next step, a rough platform design is used to map the FBs to different devices. Additional FBs are added for middleware communication between the devices. Their activation is either periodic or event-driven. A typical choice for the period of a middleware FB is the period of the data producer or the data consumer.

For example, FB4.1 is activated with the consumer period such that the undersampling between FB4 and FB7 already occurs in the sender device, thus reducing the overall network traffic. Requirements with respect to data versioning also affect the choice of the period since it has to be ensured that all required samples are actually sent.
An example for a cause–effect chain with event triggering is FB2 $\prec$ FB5 $\prec$ FB7, where FB2 is activated periodically but FB5 and FB7 are event-driven. The activation pattern of this chain reflects the pipelining used in the image processing.

An acceptable delay for each middleware communication can be estimated in two ways. On the one hand, the platform model might already indicate where low-latency communication between the devices is possible and where larger latencies have to be expected. On the other hand, the end-to-end requirement for each cause–effect chain can be compared to the latency results of the preceding functional simulation which excluded communication. As an example, it might not even be possible to formally guarantee a worst case latency for the image-processing pipeline in the red cause–effect chain. Nevertheless, the later part of the cause–effect chain is more deterministic and a solution might be that from the end-to-end requirement of 120 ms, 80 ms can be assumed for the image-processing part.
Since the first transformation is mostly a refinement of the model, it is still manageable in the ecosystem of the function developer. Consequently, simulations can be redone incorporating the new assumptions about communication delays.
Second transformation: In the subsequent transformation, an SL LET model is derived. Devices are mapped to time zones admitting a bounded synchronization error between their local clocks. An FB can be translated to an LET task; typically the period is adopted and the LET is chosen equal to the LET of the block to reflect the semantics of the functional model. Often the period is equal to the LET. A middleware FB can be modeled by an interconnect task which also inherits the period of the FB and adopts the delay of the FB as LET which may be longer than the period depending on the maximum duration of a communication.
Moreover, SL LET is able to deal with cause–effect chains which include event-triggered network communication. In this example, the interconnect task $\Phi_{5.1}$ models the production of new samples, network transport, and object detection. The task $\Phi_{5.1}$ has a period of 25 ms (the sampling frequency of the camera) and an LET of 80 ms. In contrast to a BET model, this allows a deterministic data flow behavior since FB7 can rely on a constant period and data age of its inputs.
In Sections IV-C2–IV-C5, we will revisit this example and show how further steps of MDD in combination with SL LET can be used to refine, adapt, trace, and monitor timing properties of the system, including separation of concerns between different stakeholders.

2) SL LET Allows to Intervene in the Synthesis Flow and Facilitates Specification of Timing Requirements: An advantage of the (SL-)LET when compared to synchronous programming is that a series of intermediate implementation models are accessible to the system designers. These SL LET models abstract both the applications and the execution platform and are helpful to explore system properties before code generation. The SL LET models are intuitive, close to the structure of a later implementation, and can be manipulated through user interaction.

3) SL LET Enables Composability and Timed Modularization: Today's MDD focuses on modularization of software functionality (see FBs in IEC61499). The goal is a division of labor among different working groups, respectively, among OEMs and suppliers. Besides intellectual property concerns, a supplier pursues to reuse its developed functionality in different projects and for different customers. Since the behavior of an FB is the result of the input to output relation but in combination with timing, the implementation must reproduce both. The latter aspect has been neglected in many existing MDD processes, but it becomes more important with high-performance integrated architectures. In fact, SL LET offers such modular, composable timing interfaces which can serve as a contract between different stakeholders. It is even possible to hierarchically assign LETs to subfunctions supporting a divide-and-conquer principle.

4) SL LET Eases Modifications and Platform Changes: SL LET facilitates the continuous evolution of systems by updates and reconfiguration: SL LET hides variations in response times and transmission times as long as the LETs are not exceeded. Thus, modifications with limited changes in the response times and transmission times, like the update of a function, are transparent and do not require a re-verification of the entire system.

The size of the LET is a deliberate design choice and allows us to increase the robustness of the system toward function updates at the expense of overhead. The robustness margin of an LET task is easy to compute being the difference between the LET and the current WCRT.

Also, the addition of a new component does not impact other components as long as their response times and transmission times can still be bounded from above by the respective LETs. Only if it is not possible to circumvent the transgression of an LET, it might be necessary to redefine a (sub)set of time budgets in consultation with the function developer.

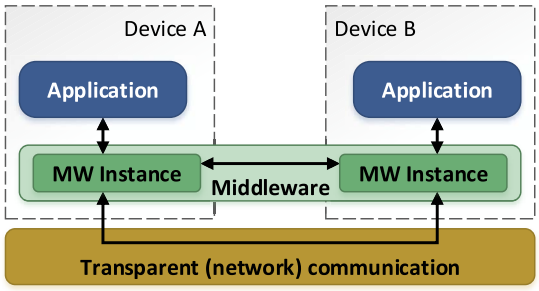


Fig. 4. ***Generic distributed system with middleware layer.***

5) SL LET Enables Efficient Implementation and Monitoring: SL LET effectively reduces the state-space of the system with regard to its RT behavior due to the introduced time determinism. This reduction in complexity can be immediately exploited in the implementation of the system. Examples are lock-free communication [31] or the reduction of memory contention [3]. Moreover, the contractual determinism of SL LET communication is a basis for monitoring which can be implemented with low minimal overhead. In Sections V and VI, we will discuss those aspects and give examples how those properties can be exploited in the design of middleware and the configuration of a network.

V. System-Level LET: Supporting the Configuration and Implementation of Middleware for RT Data Distribution

Industrial CPSs are of distributed nature: embedded controllers in sensors, machines, actuators, and so on are independent nodes in different physical locations but they must co-operate on the factory floor or in a process plant to achieve a common goal. In the future systems, even remote databases and data-processing services are accessible by connections to external servers. To facilitate the exchange of information, distributed CPSs typically feature a middleware layer as illustrated in Fig. 4 which establishes and manages the transparent distribution of data and ensures its consistent representation. By hiding heterogeneity and distribution to the individual applications, the middleware eases significantly the design and reuse of applications. A particular challenge for middleware in safety critical environments is that the communication service should fulfill soft or hard RT guarantees.

In this section, we discuss how SL LET can help to efficiently configure and implement an RT distribution middleware using the example of the DDS. Moreover, we show how SL LET can be used to manage versions of data samples in decentralized data caches.

A. Data Distribution Service

DDS is a scalable RT distribution middleware which is well established in a wide range of applications, including both industrial and automotive distributed systems [41]. As shown in Fig. 5, it consists of two layers: the upper layer is defined in the DDS standard [42] specifying a DCPS communication with the respective application interface and QoS parameters. The lower layer, standardized as the DDSI [43], defines an RT-capable middleware protocol which sets the rules for message formats and message exchange between applications.

1) Data-Centric Publish–Subscribe Model: The DDS follows a DCPS paradigm, that is, a global data space exists to which data objects can be added or from which data objects can be retrieved. In DDS, data objects are called topics, and successive values for a topic are DataSamples. An example of a topic is a sensor output which is updated periodically, leading to a series of DataSamples over time. In the following, we use the notation θ_u for a topic and the notation $\theta_{u,q}$ for its qth DataSample.

All applications that can communicate with each other are grouped in a domain, an individual application is a DomainParticipant. Each DomainParticipant has a publisher entity, consisting of several DataWriters, and a subscriber entity, consisting of several DataReaders (see Fig. 5). A DataWriter is topic-specific and communicates new DataSamples for its topic to the associated publisher. The publisher then disseminates the data values to all interested subscribers of other DomainParticipants. The receiving subscribers forward the data values to the respective topic-specific DataReaders.

Each of the DCPS entities DomainParticipant, DataWriter, DataReader, publisher, subscriber, and topic has a set of QosPolicies to specify its requirements with regard to data availability, data delivery, data timeliness, resources, and configuration [44]. Moreover, a DCPS entity has also one or multiple listener interfaces to be informed of incoming data or notifications in a synchronous or asynchronous manner.

Finally, DDS assumes that a discovery mechanism is available which: 1) detects the presence or absence of DomainParticipants and the respective DataWriters and DataReaders in the system and 2) enables negotiation between DomainParticipants deciding whether publisher–subscriber relations for common topics will be established.

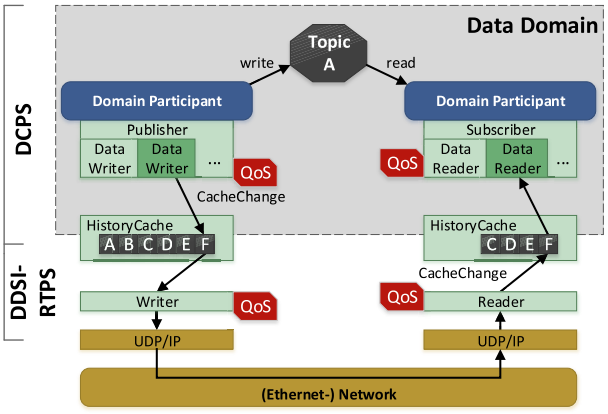


Fig. 5. ***DDS stack.***

2) Quality of Service: A distributed CPS has to maintain an internal model of the state of its physical environment which has to be sufficiently accurate. Accuracy can relate to different aspects, but we focus here on sufficient temporal consistency. Data samples which describe the state of some entity in the physical environment must be regularly updated, so that all participants have the same view on the environment. The requirement of temporal consistency can be reflected in DDS by choosing the appropriate QoS policies with regard to data availability and timeliness, which are the following.

1) **`HISTORY:`** Specifies how many DataSamples are to be cached at the DataWriter until they are delivered (resp. at the DataReader until they are taken by the application).
2) **`LIFESPAN:`** Specifies how long a sample that is produced by a DataWriter remains valid.
3) **`DEADLINE:`** Specifies the relative deadline for the update of a data sample at the DataWriter (resp. the DataReader).
4) **`LATENCY_BUDGET:`** Specifies the deadline for a message transmission, which notifies the subscribing application of a CacheChange.

An RT distribution middleware can handle QoS demands of an application in two fundamentally different ways: either it promises to do its best to satisfy the requested QoS (best-effort QoS) or it negotiates a QoS contract to which it adheres (guaranteed QoS). If best-effort QoS applies, then the middleware attempts to achieve the requested QoS by configuration and monitors the actually offered QoS at run time notifying the application in case of a failure. If guaranteed QoS applies, then the middleware checks whether a QoS contract can be fulfilled and possibly renegotiates parameters. Such contracting can be realized offline at design time or online as a reconfiguration. Regardless of how the contracting is realized, if guarantees are to be given, then a system model and an appropriate analysis must be available.

DDS applies the approach which offers best-effort QoS for a topic. This is a good approach for applications with soft RT requirements but unsatisfactory for those with hard RT requirements. Therefore, Tijero and Gutiérrez [45] and Pérez and Gutiérrez [46] explored how a system model for DDS can be built which could then be subject to classical schedulability analysis. Building such a model requires, however, knowledge about the underlying software layers and the execution platform, which must of course both have a predictable RT behavior. This will be addressed in Section V-B.

3) DDSI-RTPS: While the DDS standard [42] defines the API for transparent communication as well as the associated QoS parameters as explained above, the DDSI standard [43] specifies the underlying publisher–subscriber protocol which is called RTPS. The RTPS is characterized by the structure of its entities, the message contents, the procedure of message exchange, and the discovery mechanism as will be detailed below. Fig. 5 shows the interplay of DDS and RTPS.

a) Structure: RTPS has its own set of entities, namely participants which are composed of a set of endpoints, that is, readers and writers, as well as endpoint-owned HistoryCaches which are subject to CacheChanges modifying the cached data. The HistoryCache represents the interface between RTPS and DDS. A DDS DataWriter posts new data in the form of a CacheChange through a HistoryCache to the RTPS Writer, that is then charged to send a message to all paired RTPS Readers. An RTPS Reader then posts the CacheChange to its HistoryCache, where the DDS DataReader can retrieve it. Therefore, the HistoryCache of a writer maintains a history of made CacheChanges, while the HistoryCache of a reader maintains a history of received CacheChanges by the paired writer.

b) Messages: An RTPS message consists of a header and a set of submessages. Beside submessages for data, there are also special control submessages for heartbeats, acknowledgments, timestamps, and so on.

c) Behavior: RTPS also specifies the rules of message exchange between writers and readers for the propagation of CacheChanges, which partly depend on the chosen QoS parameters. The DDSI standard also provides stateless and stateful reference implementations of readers and writers.

d) Discovery: A PDP and an EDP must be part of RTPS. The purpose of the PDP is to identify all participants and their properties. The EDP enables communication between two participants to discover the respective writers and readers. The discovery also ensures that only endpoints with matching QoS parameters are connected.

B. Combining DDS and SL LET

The challenge in realizing a middleware for RT data distribution is to find an efficient implementation which ensures the specified QoS guarantees. In this section, we show how SL LET programming abstraction can be used for this purpose by the example of DDS. In particular, we show how the DDS entities can be mapped to the SL LET entities with a special focus on the HistoryCache, and how the existing QoS parameters can be used to configure SL LET-based DDS implementation.

1) Mapping DDS Entities to SL LET Entities: All DomainParticipants which run on the same device in the system belong to the same time zone. A time synchronization protocol, for example, PTP, is used to bound the synchronization error between the clocks of the different devices.

On each local device, the DomainParticipants are programmed such that their access to a HistoryCache through the respective DataWriters and DataReaders follows the LET programming abstraction. A DomainParticipant thus writes with a period T_λ and the logical execution time LET_λ through the DataWriter to the HistoryCache (resp. reads with a period T_λ through the DataReader from the HistoryCache). The SL LET parameters can be easily mapped to the QoS policies **`DEADLINE`**

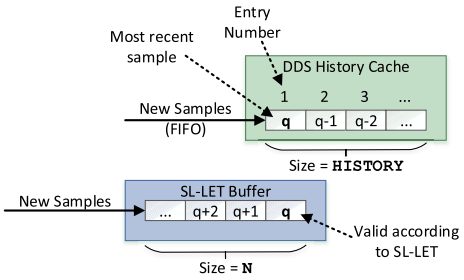


Fig. 6. ***Comparison between SL LET buffer and DDS HistoryCache.***

and **LATENCY_BUDGET** in DDS: as mentioned above, the **DEADLINE** parameter specifies how often a DataWriter must update a topic in the HistoryCache (resp. how often a new DataSample must be supplied in the HistoryCache of the paired DataReader). In a valid SL LET-based DDS implementation, the period of reading and writing data must be smaller than or equal to the **DEADLINE**. We choose here T_λ = **DEADLINE**. The **LATENCY_BUDGET** specifies a deadline for the message transmission of a DataSample, such that the LET_λ of the DomainParticipant can be as long as **LATENCY_BUDGET**.

The communication between a pair of HistoryCaches is specified by DDSI and can be mapped to SL LET interconnect tasks. An SL LET interconnect task Φ models the entire RTPS message transmission including the actions of the writer at the sending participant, the network transmission, and the actions of the reader at the receiving participant. While the I/O-behavior of Φ is thus deterministic with parameters T_Φ and LET_Φ, it allows any underlying network implementation if an appropriate buffering scheme is applied [8]. The period T_Φ must equal to the period of the sending DomainParticipant, thus here $T_\Phi = T_\lambda =$ **DEADLINE**. The LET_Φ must be chosen smaller than or equal to the **LATENCY_BUDGET**. In the following, we assume that $\mathrm{LET}_\Phi =$ **LATENCY_BUDGET**. Finally, the release offset of LET_Φ must equal LET_λ.

Note that the direct mapping of DDS's QoS policies and SL LET parameters allows us to an immediate traceability between the middleware specification and the related implementation.

2) DDS HistoryCache and SL LET Buffering: The HistoryCache, which acts as an interface between the DCPS and RTPS layers in DDS, is closely related to the buffering scheme used in SL LET [8]. Fig. 6 compares the two buffering concepts: the DDS HistoryCache is used to preserve a configurable number of DataSamples. It thus stores the most recent received sample of the topic $\theta_{p,q}$ and older versions $\theta_{p,q-1}, \theta_{p,q-2}, \ldots$—in total as many samples as specified by the QoS policy **HISTORY**. On the other hand, the buffering in SL LET prevents early arriving samples from being published before the LET_Φ of the interconnect task Φ is elapsed. Therefore, it provides N buffer entries (as depicted in Section IV-B) to store and reorder the samples, until a sample $\theta_{u,q}$ is valid according to SL LET semantics. Combining both approaches leads to a DDS HistoryCache that follows SL LET semantics.

SL LET-based DDS HistoryCache is created, if the SL LET buffer stores not only the current DataSample $\theta_{p,q}$ and the future not yet published DataSamples $\theta_{p,q+1}, \theta_{p,q+2}, \ldots$, but additionally preserves $M - 1 =$ **HISTORY** $- 1$ old DataSamples. By using $t^{Z_b}_{\mathrm{outdate}}(q) = t^{Z_b}_{\mathrm{publ}}(q+M)$, the buffer size can be calculated in the same way as proposed in [8] leading to

$$\bar{N} = M + \left\lceil \frac{\mathrm{LET}_\Phi - \mathrm{BCRT} + \Delta_R + \epsilon}{T} \right\rceil . \tag{3}$$

This can also be expressed with QoS parameters in DDS, if the above DDS-to-SL LET mapping of parameters is assumed

$$\bar{N} = \textbf{HISTORY} + \left\lceil \frac{\textbf{LATENCY\_BUDGET}_\Phi - \mathrm{BCRT} + \Delta_R + \epsilon}{\textbf{DEADLINE}} \right\rceil . \tag{4}$$

The **LIFESPAN** parameter is closely related to the concept of the HistoryCache, specifying the maximum time during which a DataSample is considered as valid. The notion of a lifetime in DDS corresponds to the concept of maximum data age in classic schedulability analysis, and it is important for function developers that need to bound the age of their input data. If DataSamples are placed at nondeterministic instants in time in the HistoryCache, each DataSample has to be annotated with a timestamp, so that DDS can iterate over the oldest samples and remove them from the HistoryCache as soon as their age exceeds the specified **LIFESPAN**. This is an expensive procedure.

Due to the strict periodic behavior of SL LET, every DataSample in the HistoryCache will always have a deterministic maximum data age. Let DataSamples be sent with the period T_Φ and the transmission delay LET_Φ, then the data age of the nth most recent DataSample ($1 \leq n \leq$ **HISTORY**) when it arrives at the HistoryCache is

$$\mathrm{min\_data\_age} = \mathrm{LET}_\Phi + (n-1) \cdot T_\Phi \tag{5}$$

and just before it becomes the $n+1$th most recent DataSample, it reaches its maximum data age

$$\mathrm{max\_data\_age} = \mathrm{LET}_\Phi + n \cdot T_\Phi . \tag{6}$$

The maximum data age of the oldest entry in the HistoryCache is thus $\mathrm{LET}_\Phi + M \cdot T_\Phi$ or in terms of DDS

$$\textbf{LIFESPAN} = \textbf{LATENCY\_BUDGET}_\Phi + \textbf{HISTORY} \cdot \textbf{DEADLINE}. \tag{7}$$

Consequently, it is up to the function design whether he specifies the number of entries that must be available in the HistoryCache (**HISTORY**) or the required **LIFESPAN** of

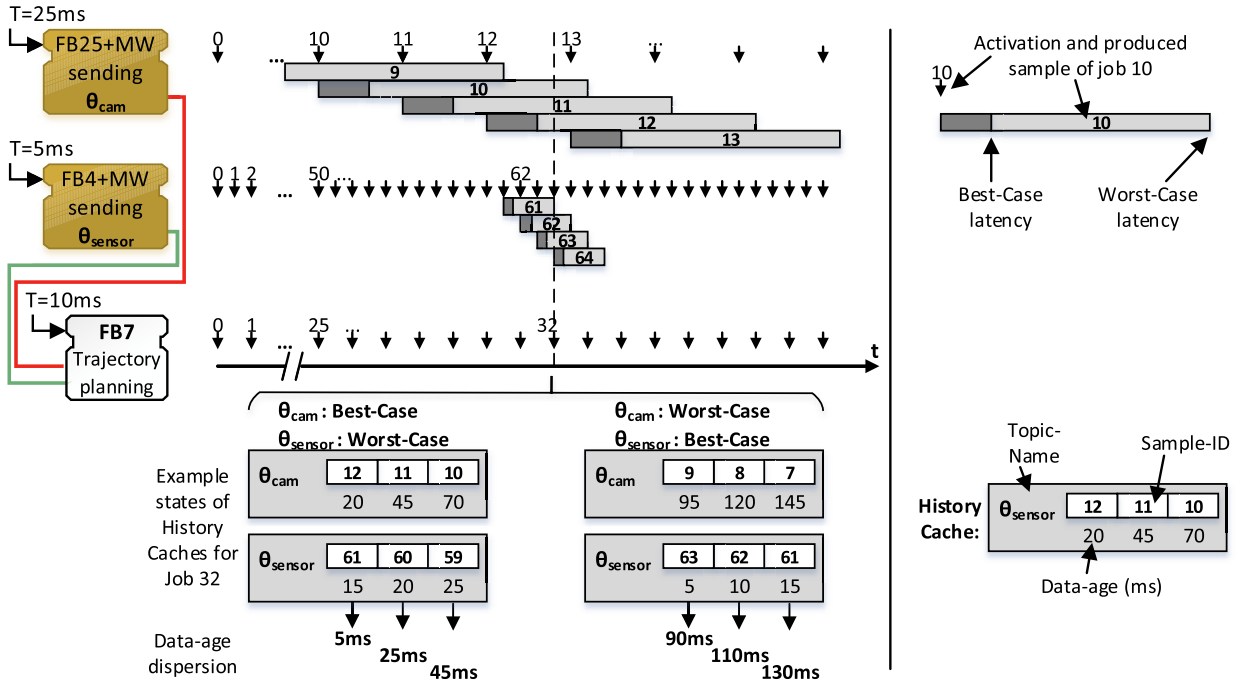


Fig. 7. ***Nondeterministic data-age dispersion when multiple inputs with BET communication are combined.***

each DataSample. It is important to mention that the lifespan of the sample is specific to the reader (resp. the receiving time zone). This flexibility is inherited from the SL LET paradigm and allows, for example, to only store the most recent sample in the sending time zone (normal LET behavior), while the output of the SL LET interconnect task has a reader-dependent lifespan in the receiving time zone.

3) Mastering Dispersion in Data Ages: We have seen that the implementation of a middleware for RT data distribution like DDS is considerably eased if an SL LET-based approach is chosen. On the one hand, mapping QoS policies to SL LET parameters is straightforward such that a correct configuration of a DDS implementation is no longer problematic. On the other hand, the original QoS requirements can be directly retrieved from the implementation such that high traceability is given. Moreover, we have shown that the nth DataSample in a HistoryCache is guaranteed to be updated with period **`DEADLINE`** and has a deterministic **`LIFESPAN`**. In the following, we will consider the scenario in which a CPS application reads DataSamples of two topics and thus from two history caches that are updated by (remote) DataWriters. A requirement is that the dispersion in data ages of the DataSamples from the two different sources is deterministic and behaves as observed in the simulation of the functional model. We will see based on an example that SL LET-based implementation for DDS, in contrast to a BET-based implementation, fulfills this requirement immediately.

Consider the trajectory planning (FB7) in the pHRI use case from Fig. 3. It has to correlate sensor data from the robot (FB4) with results from the camera-based object detection (FB2, FB5). While the closely coupled robot sensors have a higher sampling frequency and typically a more deterministic data transport (low jitter), the complex image processing and network transport leads to a higher jitter in the red cause–effect chain. The jitter in the transmission of DataSamples to the HistoryCaches is not part of the functional model which assumes that the middleware contributes a static delay.

a) BET-Based Implementation: Fig. 7 shows how a BET-based communication between FB2+5, FB4, and FB7 might look like. We assume that all FBs have a strict period and their schedules are started synchronously with activation of job 0.

1) For simplicity, we model the beginning of the red cause–effect chain (FB2, MW 2.1, FB5, MW 5.1) with a single FB2+5+MW that produces samples with a period of 25 ms. Let the execution of FB2+5 be subject to a large jitter such that it might take between 15 and 80 ms from the recording of a camera image until the processed data sample is received by FB7. The variable part of the latency is represented by a light gray box in Fig. 7.
2) Also for simplicity, we summarize FB4 with an MW 4.1 to a single FB4+MW, which has a five times shorter period than FB2. We assume a low jitter for the transmissions between FB4 and FB7.
3) We further restrict the example to pipelining, such that DataSamples always arrive in order.

However, different routing paths in a real-world scenario might result in out-of-order arrivals. This would lead to an even higher complexity here, as the entry of a sample in the history cache would not necessarily correlate with its data age. Let FB7 store the three most recent samples

of each topic in its HistoryCaches (**HISTORY** $= 3$), they can be used by FB7 to calculate the motion of detected objects. The topics are named by their producer FBs. Snapshots of the HistoryCaches at the 32nd activation of the consumer block FB7 are shown in the lower part of Fig. 7. The first snapshot shows a corner case in which data is produced and transmitted with minimum latencies for the red cause–effect chain but with maximum latencies for the green cause–effect chain, such that $\theta_{\text{cam},10}$, $\theta_{\text{cam},11}$, $\theta_{\text{cam},12}$, and $\theta_{\text{sen},59}$, and $\theta_{\text{sen},60}$, $\theta_{\text{sen},61}$ are available at the moment when FB7 reads. The second snapshot shows another corner case: here data are produced and transmitted with minimum latencies for the green cause–effect chain but with maximum latencies for the red cause–effect chain, such that $\theta_{\text{cam},7}$, $\theta_{\text{cam},8}$, $\theta_{\text{cam},9}$ and $\theta_{\text{sen},61}$, and $\theta_{\text{sen},62}$, $\theta_{\text{sen},63}$ are available at the moment when FB7 reads. The entries of the HistoryCaches are also annotated with their current data age.

The comparison of the two corner cases reveals a problem that the function developer of FB2 is faced with. Although it is possible to specify the **DEADLINE** and **LATENCY_BUDGET** parameters and they can be fulfilled by a given BET implementation, there is no way to impose and realize a deterministic data-age dispersion. Therefore, the function developer has to implement FB7 with vague assumptions about the correlation of the data ages in the two HistoryCaches. Remember, however, that the two inputs represent detected objects from the camera as well as sensor data from the robot and a correct trajectory computation is safety critical.

b) SL LET-Based implementation: By implementing the above example with SL LET, a deterministic run time behavior is enforced as shown in Fig. 8. The LET for the block FB2+5+MW (resp. FB4+MW) is set to the sum of LETs of the subblocks and the **LATENCY_BUDGET**(s) for the middleware. The implementation behavior corresponds consequently to the simulated, deterministic behavior of the functional model: for instance, there is only one possible state of the HistoryCaches in the example when the 32nd job of FB7 is activated as illustrated in Fig. 7. It is interesting to see that the dispersion in data ages is even lower than observed in the BET setup due to the reduced relative variability.

To cover the general case, let us now formally derive the data-age dispersion of two DataSamples from topics θ_p and θ_k which are read in simultaneously by a receiving application FB_j under the assumption of SL LET-based implementation of DDS. To begin with, we determine which DataSample of topic θ_p is in position $n = 1$ in the HistoryCache H_p at the activation instant of ith job of FB_j. Clearly, the most recent available DataSample of topic θ_p in the HistoryCache H_p must be $\theta_{p,q}$ with

$$\begin{aligned} q &= \left\{\max\ q' \mid i \cdot T_j \geq q' \cdot T_p + \text{LET}_p\right\} \\ &= \left\lfloor \frac{i \cdot T_j - \text{LET}_p}{T_p} \right\rfloor \cdot T_p. \end{aligned} \tag{8}$$

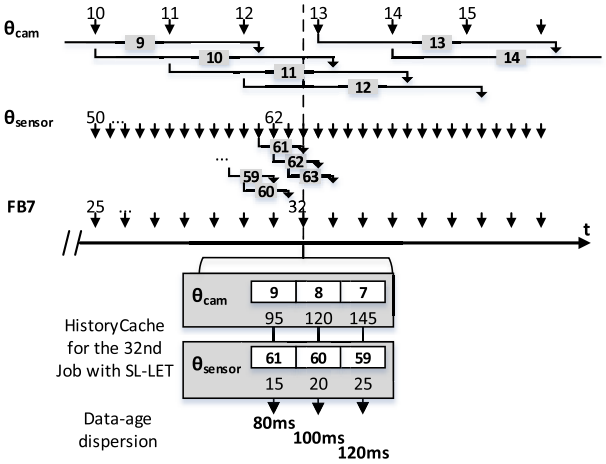


Fig. 8. ***Data-age dispersion with SL LET.***

At the moment when the ith job of the consumer FB_j reads the DataSample $\theta_{p,q}$ in position $n = 1$ of its HistoryCache, it has the data age

$$\begin{aligned} &\text{data\_age}(\theta_p \xrightarrow{1} \text{FB}_j(i)) \\ &\quad = i \cdot T_j - q \cdot T_p \\ &\quad = i \cdot T_j - \left\lfloor \frac{i \cdot T_j - \text{LET}_p}{T_p} \right\rfloor \cdot T_p. \end{aligned} \tag{9}$$

In the general case, when the ith job of the consumer FB_j reads the DataSample $\theta_{p,q}$ in position n of its HistoryCache with $1 \leq n \leq$ **HISTORY**, the respective data age is

$$\begin{aligned} &\text{data\_age}(\theta_p \xrightarrow{n} \text{FB}_j(i)) \\ &\quad = i \cdot T_j - \left\lfloor \frac{i \cdot T_j - \text{LET}_p}{T_p} \right\rfloor \cdot T_p + (n-1) \cdot T_p. \end{aligned} \tag{10}$$

In other words, the data age of the DataSample $\theta_{p,q}$ in position n is always the same for the ith job of FB_j in any execution run of the implementation. This immediately leads to a constant dispersion in data age when the ith job of FB_j reads a DataSample $\theta_{p,q}$ in position n and a DataSample $\theta_{k,l}$ in position m:

$$\begin{aligned} &\text{data\_dispersion}\left(\theta_p, \theta_k \xrightarrow{m,n} \text{FB}_j(i)\right) \\ &\quad \left|\text{data\_age}(p \xrightarrow{n} \text{FB}_j(i)) - \text{data\_age}\left(k \xrightarrow{m} \text{FB}_j(i)\right)\right|. \end{aligned} \tag{11}$$

In conclusion, we can say that SL LET enables a fully time-deterministic management of data versions both with regard to absolute data age of DataSamples and the relative data ages of DataSamples from different topics.

4) Monitoring QoS Parameters in SL LET: The determinism in an SL LET-based middleware enables an elegant implementation of run-time monitoring. This is especially relevant for complex CPSs where the timing behavior cannot be formally proved during design time. Therefore,

adherence to the QoS parameters has to be validated during operation. Different options are possible in the case of a violation, which can either ensure that the system enters a safe state (fail-safe) or that operation can continue in a safe way (safe-operational). DDS provides the ability to inform the application about a violated QoS parameter, such that an appropriate reaction can be implemented by the function developer. On the other hand, it would also be possible to relieve the application from that burden by integrating a reaction in the middleware itself.

An important monitoring case for the discussed safety-relevant systems is the missing of an expected sample when the receiving job is activated. This might occur either due to an unpredictable large delay in the preceding part of the cause–effect chain or due to packet loss in the network. Both cases would modify the data flow in the cause–effect chain and could lead to a behavior that does not correspond to the specification. Without SL LET, this can only be implicitly detected by monitoring the interarrival times of the samples. Nevertheless, the absence of a packet can only be detected by continuously setting a monitoring event with a deadline relative to the last received sample, which equals a watchdog timer. Timestamps can also be used to monitor the latency of each transmission. This already requires time synchronization between the sender and the receiver and an exceeded latency budget can only be recognized retrospectively.

By introducing SL LET, monitoring decisions can be made with lower overhead. As shown above, **`DEADLINE`** and **`LATENCY_BUDGET`** are used to describe SL LET-based communication in DDS. Since both participants have sufficient time synchronization, an explicit deadline for each sample exists in the time zone of the receiver. This deadline equals the LET mark for publishing the sample. Consequently, it is possible to set periodic monitoring events to the points in time when a new sample is to be published in the receiver time zone. A special case for the application of this monitoring is the image processing at the beginning of the red cause–effect chain in the pHRI use case (see Fig. 7). While the image processing includes event-driven functions with pipelining, it can be abstracted by an interconnect task. The input for the trajectory planning (FB7) can therefore rely on a constant period of new samples (frame capture rate of the camera) and a constant data age due to the abstraction of delay with the interconnect tasks LET. Any form of misbehavior can therefore be detected.

Based on the reliability requirements, multiple reactions are conceivable. On the one hand, the application may simply be informed about the missing sample, which can also be done by delivering a predefined error value when the sample is requested. In the case of fail-operational systems, however, redundancy may be required. SL LET simplifies the voting between multiple redundant inputs like the time instant for voting is explicitly given by the elapse of the interconnect task's LET. At the time of voting, one has to check whether the primary sample is available or if the sample from the redundant path has to be used. This can be done seamlessly by the middleware, such that the application is not affected at all.

VI. System-Level LET: Supporting the Configuration and Implementation of TSN

TSN comprises a set of standards that are designed for timeliness and reliable communication over Ethernet [47]. In Section VI-A, we will outline the relevant TSN standards and their evolution. Based on that, Section VI-B shows which challenges occur when TSN standards are deployed in industrial automation systems and how SL LET can be used to improve the design process regarding compositionality and platform independence.

A. Overview on TSN Standards

The first evolution of industrial and automotive networks was dominated by the introduction of fieldbuses as, for instance, Profibus, Foundation Fieldbus, CAN, or FlexRay [48], [49]. This paved the way for the first distributed applications while reducing cabling costs and weight. Meanwhile, Ethernet-based communication had massive success in general-purpose computing due to the unified communication backend, an extensible protocol stack, and open standards.

Unfortunately, the missing time predictability of classical Ethernet required additional effort, resulting in new industrial Ethernet variations like EtherCAT [50] or Profinet [51]. However, such protocols support IP-based communication only as an option for non-time-critical communication and therefore again require a specialized protocol stack and individual addressing. As part of the fourth industrial revolution as well as in the context of automated driving, the need for IP-based communication that stays closer to the classical Ethernet standards arose. This was mainly driven by the motivation of removing unnecessary gateway complexity between different levels of hierarchy (e.g., including applications in the cloud for computation-intensive tasks or data aggregation). In the domain of audio and video broadcasting, the TSN Task Group was established to develop a new set of standards providing QoS guarantees for Ethernet which go beyond the eight priority classes defined in 802.1Q [52].

The set of TSN standards can be split into three major topics, namely time synchronization, scheduling, and traffic control. In the following, we will highlight the relevant standards and their relation to industrial automation as well as in-vehicle communication.

1) Time Synchronization: First, RT systems usually require a common notion of time that has to be shared by different system parts. For consumer electronics and web services, the NTP provides a master–slave time synchronization with an accuracy of a few milliseconds. In contrast to this, factory automation imposes higher requirements on clock synchronization. This includes a worst case

deviation of clocks in the order of tenth to a few hundred nanoseconds as well as robustness against a failure of the master clock. The PTP improves over NTP by providing a higher accuracy even down to a few nanoseconds (IEEE 1588 [36]) and zero failover times with multiple master clocks are possible (IEEE 802.1AS-rev [53]).

2) Scheduling: The second group of TSN standards comprises scheduling and traffic shaping mechanisms to ensure deterministic and low latencies for packet transfer. Most notable is the 802.1Qbv TAS that specifies a TDMA scheduling for strict temporal isolation of Ethernet traffic. The TAS consists of two hierarchical levels. In the first stage, a cyclic gate scheduler decides which priorities (separated in different queues) are taken into account for each TDMA slot. The second stage is the classic priority scheduler according to 802.Q that selects the packet with the highest priority for transfer. The cyclic behavior is convenient for both, the automotive and the industrial domain, since it is able to imitate the schedules of established buses like FlexRay and Profibus and fits well for periodic control applications.

802.1Qch aims to provide short end-to-end latencies by synchronizing the TDMA slots on different network links. The goal is to prevent long stall times that occur if a packet arrives at the switch right when its corresponding TDMA slot for next network link is over. Therefore, 802.1Qch proposes a synchronization of the schedules for all links on the critical path. This is closely related to the concept of IRT communication in ProfiNet (RT_CLASS_3), where intermediate buffering is eliminated [48].

One restriction implied by the TDMA schedule is the guard band at the end of each TDMA slot. It ensures that the last packet in the slot is not able to reach the subsequent slot, enforcing thus the strict temporal isolation. Therefore, during the guard band, whose size equals a maximum-sized Ethernet packet (1534 Bytes usually), no new transmission is allowed resulting in a bandwidth reduction. To limit this problem, 802.1Qbu specifies frame preemption, where a running transmission can be preempted once by a higher priority frame [54]. This reduces the size of the guard band to the size of the smallest possible preemption fragment (64 Bytes).

3) Traffic Control: The third group of TSN standards addresses the topics of traffic path control and reliability. Path control and reservation as specified in 802.1Qca are closely related to the domain of SDN. Besides shortest path forwarding, it allows us to specify explicit trees for managing traffic paths. It also allows us to reconfigure those rules during run time to adapt the network to system changes like exceptions or updates.

Finally, 802.1CB specifies FRER which enables seamless redundancy by replicating frames in the network switches. Therefore, the clients should be relieved from the burden of ensuring reliability on the application level.

In conclusion, TSN provides a large set of standards, targeting time-predictable and reliable Ethernet communication. Unfortunately, this results in complex design space with a variety of configuration parameters for each standard [54]. To assist here, multiple TSN profiles are currently under development proposing guidelines and standard configurations for the deployment of TSN in different domains [47]. Examples are IEC/IEEE 60802 for industrial automation and IEEE 802.1DG for in-vehicle Ethernet communication.

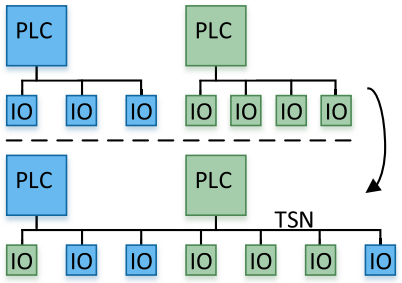


Fig. 9. ***Coexistence between two brownfield systems with TSN.***

In the following, we will show how SL LET can be used beneficially here to enable deterministic timing and therefore compositionality.

B. Enabling Compositionality of TSN Networks With SL LET

One of the most important aspects of the fourth industrial revolution is the interoperability of all hierarchical layers [38]. Besides the hierarchy of a factory (see Section II) with its different communication technologies, this also covers organizational structures including stakeholders in application design, verification, and integration. Obviously, it is not affordable to simply replace existing structures and start from scratch (also known as greenfield development). Instead, a set of legacy hardware and the software will be preserved (also called brownfield) and needs to be taken into account.

Coexistence, as the first level of interoperability, means that network and stream configuration is done in such a way that the requirements of all existing stakeholders are fulfilled. An example as shown in Fig. 9 is a setup with brownfield devices from different vendors that should be connected to a central TSN network. Therefore, a TDMA schedule for the TAS needs to be implemented such that the existing requirements, for example, with regard to the network cycle time and their minimal TDMA slot size are fulfilled, ensuring temporal and spatial isolation of their communication. Computing such a TDMA schedule is an optimization problem that has to be solved for the whole network [54], [55].

The next reasonable level of interoperability is co-operation. It means that existing devices as well as new devices have to be connected to fulfill a common task. In contrast to coexistence, which focuses on the lower layers of the OSI model (physical, data link, and network layer), co-operation also affects the upper layers such as the applications. This requires a unified communication

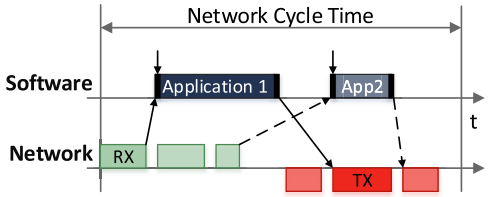


Fig. 10. ***Tight coupling of network and application scheduling for short latencies.***

interface between the devices while each vendor still has to preserve its intellectual property.

In the following, an example is discussed how such a co-operating system can be integrated at the network level: The TDMA cycle of the TAS is a property that affects all connected participants. Formulated as an optimization problem, a TDMA slot can be reserved for each message that has to be transferred and the execution of applications can be synchronized in order to achieve minimal latencies. Fig. 10 illustrates a scenario in which the activation of application 1 is synchronized to the arrival of input data. As a result, network and application design as well as software integration are coupled tightly together. One might even state that "the system is built around the network."

Although a system-wide optimization is able to achieve the shortest possible latencies, there is a set of substantial drawbacks connected to this approach of system design: Global optimization prevents the division of labor between network and application design as well as software integration. Typically those tasks would be done by different teams and even different vendors. In particular, the execution period of control applications should be rooted in the design of the controller and not be affected by the network design or integration. The optimization results show low stability, as only small changes in the input data set might result in a very different output configuration. Consequently, changes and reuse of software are made difficult.

The discussion reveals that the major issue of the existing design process is the scheduling in TSN. Although the end-to-end deadlines of each communication path are part of the optimization problem [55], the timing is only deterministic for one specific output configuration. This is because the latency constraints only represent upper bounds. While the TAS enables low jitter, a solution to the optimization problem might lead to a latency that is smaller than the constraint. Consequently, this does not necessarily represent the behavior specified in the functional model. Moreover, with each reconfiguration of the network, the actual latency varies although the overall constraint may still be fulfilled. This might lead to a modification of the data flow in all cause–effect chains, even though they are not directly affected by the update.

Fig. 11 shows an example where the message from the sender is mapped to a TDMA slot such that the latency constraint is fulfilled (solid lines). The resulting dependence between the sender and the receiver jobs is denoted by the colors of the jobs. Assuming that the optimization is repeated, for example, due to an update in the system, the messages from the sender might be mapped to a different TDMA slot. Although the latency constraint is still fulfilled, this results in a different data flow that is not detected by the function developer (dashed lines). This is of particular importance when the reader uses data versioning and combines inputs from multiple writers as discussed in Section V.

SL LET can be integrated seamlessly in this process. The specified LETs for the interconnect tasks can be used as constraints in the optimization. Nevertheless, SL LET does not imply any restrictions on the underlying scheduling. If very short latencies are achieved by an optimized TAS configuration, this can be abstracted by the interconnect task as well as a heterogeneous network with larger latencies. SL LET enables compositionality here since the data are published to the applications independent of the exact configuration of the network. The only restriction is that the worst case latency constraints are met. If the global scheduling is recalculated and the latency of one cause–effect chain is smaller than before, this is not visible to the applications and does not affect the behavior of the entire cause–effect chain.

VII. EVALUATION

In this section, we evaluate SL LET in the context of industrial CPSs. In Section VII-A, we show that SL LET satisfies important properties of an RT programming paradigm for heterogeneous system integration which were first stated in Section I. Section VII-B compares SL LET to synchronous systems [14], LET [2], and the reactor concept [12]. Finally, we investigate the implementation and runtime overhead of SL LET in Section VII-C.

A. SL LET in the Context of Industrial CPS

SL LET, as an RT programming paradigm for industrial CPS and thus safety-critical systems, must enable a synthesis process that results in correct implementation of the given functional model. We show that because SL LET complies with the requirements from Section II, namely time predictability and data flow determinism as well as composability and platform independence, correct implementation can be efficiently realized.

1) Time Predictability and Data Flow Determinism: The implementation process typically starts from a function

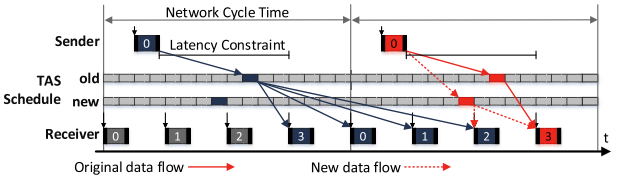


Fig. 11. ***Modified data flow although optimization constraint is still fulfilled.***

model described in terms of the standard IEC 61499. A correct implementation has to reflect the model in terms of an unambiguous behavior during runtime. SL LET is beneficial here, as it inherits the time predictability and data flow determinism from LET and extends its applicability to distributed systems. Time predictability is ensured, since the SL LET interconnect task reads and writes at defined time instants and therefore behaves like an LET task in the source and destination time zone. In combination with the time synchronization between both time zones, this ensures a predictable timing behavior for the communication path covered by the interconnect task. Just like it is done for LET in the local case, a response time analysis can be performed for SL LET, covering everything that is abstracted by the interconnect task.

It is important that SL LET is not restricted to specific scheduling on the communication network, as long as an upper bound for the latency can be computed. SL LET especially allows us to incorporate event-triggered parts in a cause–effect chain, as long as they have a time-triggered start and endpoint. This is highly relevant for distributed CPSs, as it enables the use of event-driven communication networks as well as abstracting processing pipelines. In particular, the latter ones likely have a latency which is much larger than the (sampling) period. This is an improvement over LET, which is limited to time-triggered cause–effect chains with implicit deadlines and is therefore not applicable here.

Exactly as with LET, the deterministic read and write instants enable data flow determinism for SL LET [8], [56]. This is a major benefit compared to BET, as it prohibits ambiguous data flow caused by latency jitter.

We have further shown in Section V-B that this is not limited to a simple register-based communication semantic. Instead, SL LET harmonizes with data versioning in state-of-the-art middleware like DDS and the provided data flow determinism allows us to tackle the synchronization challenge when it comes to crossing cause-effect chains.

2) Composability and Platform Independence: The trend toward higher flexibility in distributed CPSs requires a design process that incorporates updates and modifications as a central aspect, not as future work. Therefore, composability and platform independence gain more importance.

With SL LET, the distinction between logical and physical timing enables composability for distributed CPSs. The data flow and timing behavior of an interconnect task is not affected by modifications (e.g., by adding a foreign interconnect task), as long as those modifications do not endanger the LET of the interconnect task. Therefore, it is favorable to include slack in the LET of the interconnect task, making it more robust to changes.

The composable timing of an SL LET interconnect task also eases monitoring of the implementation during runtime. This is an important safety precaution for a CPS that physically interacts with a human, where any abnormal timing behavior in a cause–effect chain may cause damage to health. With SL LET, each data sample has an explicit, yet simple, deadline when it must be produced and when a transmission has to be completed. Reasons for a timing violation might be an unexpected network overload as well as a software or hardware failure within the cause–effect chain. The proper reaction to a timing violation then depends on the use case. It may range from a notification, over an emergency stop up to the switch over to a redundant channel.

In addition to composability, SL LET is also platform-independent in a way that mapping an SL LET interconnect task to a different hardware platform does not alter the data flow or timing behavior of the interconnect task, as long as the new mapping can still meet the latency constraint imposed by the LET. The reason for such a mapping decision might be that the producer and/or the consumer of the data has to be migrated to a different processing resource or that the underlying communication medium is modified (e.g., replacing a fieldbus with a TSN network).

The component-based distinction between logical and physical timing behavior in SL LET makes the complex synthesis process manageable for the designer. This is important when a CPS consists of a variety of different functions from different development teams/vendors. SL LET enables a component-based synthesis process here, where each participant can have an in-depth knowledge of its own synthesis results while abstracting foreign components only by their timing specification (a type of contracting). It further allows us to modify only parts of a cause–effect chain without touching the remaining implementation. As shown in Section VI-B, this can also be used to provide an abstraction for a TSN network, where the data flow is preserved although each modification results in a new TDMA slot assignment. This is an important aspect keeping in mind that network and application design are two candidates likely being separated into different development teams.

In summary, the main benefit of utilizing SL LET in industrial CPSs is its versatility in different stages of the design process. SL LET can act as a paradigm for specification of timing requirements but also allows efficient implementation and is applicable to existing communication and data versioning techniques like DDS and TSN.

B. SL LET Compared to Other Approaches

We have discussed SL LET and its applicability to distributed CPS for industrial automation. However, there exist other RT programming paradigms as shown in Section III, for which we will give a short comparison with SL LET. The LET paradigm is well known and standardized in the automotive domain [9], but it is restricted to periodic tasks (time triggering) with implicit deadlines as well as a register semantic [2]. Although this was sufficient to adapt control applications from single to shared-memory multicore platforms [31], SL LET has been developed to

extend the approach to larger NUMA architectures and distributed CPS. The main differences are that SL LET relaxes synchronous scheduling to tightly couples LET "islands" (time zones) with their own local schedule and allows to incorporate event triggered parts in a cause–effect chain, for example, for communication networks as well as processing pipelines (see Section IV-C). In this article, we have shown that SL LET is not only applicable to register communication, but also allows us to implement publisher subscriber systems with a more complex data versioning (see Section V-B). We used DDS as one example for a widely used RT middleware in robotics, but SL LET should be applicable to any other middleware. The advantages of combining data versioning with buffering, however, will only be effective, if that middleware provides data versioning or at least a history buffer. Efficient data caching mechanisms would be very helpful.

As an alternative, synchronous systems are originally based on event triggering and can be extended to periodic activation. However, covering asynchronous networks in distributed CPS is challenging though different approaches have been proposed [20]–[25]. Also division of labor in a synthesis process is hard to achieve. Implementing parallelism in distributed CPSs is still a complex task and a couple of works exist for parallel execution of synchronous programs [19]. SL LET allows us to explicitly incorporate a bounded synchronization error between the distributed CPS running in parallel. We believe that the SL LET paradigm could also be applied in the extension of synchronous systems.

The reactor concept is relatively new and focuses also on event-driven systems, while time is considered in the form of physical and logical timestamps [32]. It has in common with SL LET that both provide data flow determinism, composability, and platform independence. To achieve data flow determinism, reactors make use of logical timestamps for reordering and correlating events [12]. To our knowledge, a general scheduling analysis for the reactor concept has not been shown yet, while SL LET builds its time predictability on top of existing scheduling analysis. SL LET therefore can be implemented without timestamp processing at the receiver, since the timing information is implicitly given by an LET scheduler [8].

C. Implementation and Runtime Overhead of SL LET

Given the host of CPS HW/SW platforms, SL LET overhead evaluation can only cover the core functions and their impact compared to existing implementations. The comparison must be general enough to be valid for a wide range of applications rather than for a single use case. For this reason, we decided to consider the implementation overhead induced by SL LET and its basis, LET, in general terms, and refer to overhead data from the literature wherever available.

Furthermore, for an impartial evaluation, SL LET is compared to an arbitrary approach that correctly implements a specified application function with deterministic data flow on a distributed platform. Any such approach must mask the effect of communication jitter on data flow. We further assume that this application is an RT problem, that is, the application has an end-to-end deadline. Any approach must guarantee that the worst case end-to-end latency (i.e., the response time) is shorter than the respective deadline. Such a guarantee requires some form of time predictability as explained in Section II. We do not require the remaining items of the problem description, composability, and platform independence. This is a contribution of SL LET.

We include three key overhead metrics in our comparison which are as follows:

1) memory overhead;
2) workload overhead;
3) worst case end-to-end latency overhead.

The first overhead metric, memory overhead, can be further classified as additional memory for management/configuration data, for buffering as well as for program code. We will discuss all three classes.

The overhead for LET management and configuration data as well as program code has been investigated before. In [4] and [57], a control application running on a popular AURIX multicore platform with an ERIKA OS caused a total additional program and configuration data requirement of ca. 1 kByte, mainly for interrupt service routines handling the data context switch at LET labels. A prototype implementation of SL LET [58] shows that the actual overhead is comparable to the overhead of a LET implementation. The main extension to LET is that SL LET uses the existing communication stack to communicate with other time zones. This stack is needed irrespective of the programming paradigm and is not counted as overhead.

With this negligible overhead, data buffering represents the more important part of memory consumption and can be subdivided into three different parts, the buffer memory induced by LET, by an application, and by SL LET. Again, we start with the LET induced overhead as part 1. In its abstract model, LET assumes ideal zero-time communication, where a sample is produced instantaneously and can be consumed at the same moment in time. This was originally one of the most important motivations that paved the way for LET, as it was exploited for lock-free communication between processor cores. In the implementation, writes and reads go to different entries of a double buffer, and only the buffer switch is executed as an effectively atomic instruction [4]. So, the double buffer storing LET variables twice is the main LET memory cost factor. That cost could be substantial for applications with large data objects, such as the pHRI use case where camera data are processed. There are ways to mitigate this memory overhead, reducing object size by communicating data in smaller chunks, and optimizing the schedule such that read and write accesses are mapped to different execution phases, as elaborated in [31]. Such schedule optimization,

however, is not for free. While preserving data flow determinism, adapting the schedule deviates from the original zero delay communication model, such that changes inside an LET scheduled domain require resynthesizing the whole schedule. That resynthesis, however, only affects a single LET scheduled domain, that is, an SL LET time zone, while all other domains are unaffected due to the interconnect task mechanism. So, even with LET schedule optimization, SL LET composability is preserved.

Part 2 is the buffer memory required by an application. It depends on the use of past data samples and imposed requirements on data-age and data-age dispersion. The history is needed in any implementation and can, therefore, not to be attributed to SL LET.

However, history buffering can be combined with SL LET buffering for early arriving data as shown above for the case of DDS. The required memory size for the early arriving data is part 3 of the buffer memory and depends on the difference of earliest and latest arrival times of data, that is, on the computation and communication jitter. Any approach that guarantees data flow determinism and availability of the respective data at their deadline must be able to provide the respective buffer space, including systems with event-based activation. A SL LET implementation does not add to this requirement. So, there is no overhead in part 3.

In summary, the memory overhead includes the small overhead for LET and SL LET management and configuration plus the larger LET buffer overhead that, however, can be mitigated by established optimization methods.

The second metric in our comparison is the workload overhead of an application. It represents the percentage of time that a platform spends on the execution of an application. RT applications rarely run alone but share the platform with non-RT tasks (diagnosis, housekeeping, etc.). The RT application load determines how much resource time is available for such non-RT tasks. In particular, non-workload-preserving scheduling strategies, such as TDMA (see TTA [59]) incur load overhead by construction. We evaluate if, instead, workload overhead can be mitigated when implementing LET and SL LET.

In general, SL LET builds on top of two approaches. On the one hand, an existing LET system, and, on the other hand, a communication backend. As discussed, the latter one includes functionality such as the middleware and a network stack and is required for any distributed CPS independent of SL LET. Therefore, the workload overhead of SL LET only depends on how the LET paradigm is implemented. Beckert and Ernst [4] have proposed a workload-preserving implementation for LET that is based on an SPP scheduler with table-driven activations. They have shown that it can be implemented with a timing and workload overhead of less than 1% for an AURIX multicore processor with ERIKA OS RTE. This figure refers only to the LET workload (otherwise it would even be lower). This is accomplished by using hardware timers to generate events for buffer handling and task activation based on a predefined schedule table. The low overhead is the more important as, at the same time, much of the workload increasing spin-locking overhead for task synchronization can be removed. This is a result of the lock-free communication explaining the increasing popularity in automotive software development.

Since the interconnect task behaves as an LET task in the sender and the receiver time zones, the solution from [4] can be reused, for example, to trigger the transmission of samples in the case of SL LET. As discussed above, the received sample can be inserted in a buffer efficiently with a modulo operation. (SL)-LET neither requires a time-driven network nor a local TDMA schedule. As a result, also SL LET can be realized with a negligible workload overhead of around 1%. This aspect is of major relevance, as the original LET implementation in the Giotto system [26], [60] resorted to a nonworkload preserving time-triggered scheduling strategy.

In summary, the workload overhead of SL LET is dominated by the LET mechanism that is as small as about 1% of a typical workload, not even counting the beneficial reduction in spin-locking cost.

The third metric in our comparison is the system response time, here the worst case end-to-end latency. This is a crucial metric for any complex CPS, because it defines the dead time in a feedback loop, such as in our pHRI use case. To protect the feedback application, end-to-end deadlines are defined to delimit the acceptable dead time. We assume a hard deadline, which is certainly the case for our motivating use case. Because only implementations with a worst case end-to end latency shorter than the deadline are permitted, the worst case end-to-end latency overhead is a key metric for schedulability.

We will, again, separate the latency overhead in LET induced timing overhead and SL LET-induced overhead. This is possible because the response times of the time zones and their interconnect add up to the end-to-end latency. We start again with LET timing before we discuss the impact of SL LET. Here, we can directly go back to the LET workload discussion above. The extra workload is caused by the buffer and task activation management that is included in any cause–effect chain. So, the 1% timing overhead also applies to the LET part of the end-to-end latency, again not considering the potential reduction due to avoided spin-locking.

The SL LET-induced overhead can, again, be divided into communication timing and synchronization error impact. SL LET can be implemented with minimal communication overhead by utilizing a ring buffer at the receiver side as explained above leading to a negligible buffer control overhead of a few clock cycles. The communication backend itself (e.g., a middleware and a network stack) is independent of SL LET and, therefore, does not add to the communication overhead. The implicit notion of time, based on the time-driven scheduler, is a key enabler for such a low communication overhead as otherwise explicit timestamps for each sample have to be processed to rearrange the samples in the receive buffer. Secondly, the LET of the interconnect task must be larger than the

WCRT of the interconnect task plus the synchronization error [8]. This error (and therefore the overhead) can be minimized to the range of 20–100 ns with PTP clock synchronization [36]. This is usually negligible. It should, however, be noted that the synchronization error must be compensated at every time zone transition. Large clock deviations will, therefore, have an unmitigated multiple impacts on end-to-end timing. Fortunately, SL LET allows us to handle synchronization errors between time zones individually. That fits well with the hierarchy of industrial systems where the lowest hierarchy levels provide the best synchronization accuracy to support short end-to-end deadlines.

There is one aspect missing so far, that is slack in the LET timing (see above). The slack adds to the end-to-end latency at every corresponding LET task along a cause–effect chain. It is not induced by SL LET and therefore is not counted as SL LET overhead, because it could be set to 0. However, slack is important as a designer-controlled parameter to reach robustness against changes. So, the designer must balance between slack against worst case end-to-end latencies. Again, this will be necessary for any implementation of data flow deterministic cause–effect chains.

In summary, the basic impact of LET and SL LET on worst case end-to-end response times and, hence, achievable deadlines is very small, but is influenced by two major factors, accuracy of clock synchronization and slack that is inserted to increase robustness.

VIII. CONCLUSION

To cope with growing computing performance requirements, CPS architectures are moving toward heterogeneous high-performance computer architectures and networks. Such architectures integrate a growing number of complex functions under RT and safety constraints. This development challenges the established incremental design style with reuse and frequent updates.

Using an industrial pHRI use case as motivational example, we argued that this development asks for a robust and predictable RT programming paradigm. For such a paradigm, we formulated four requirements of time predictability, data flow determinism, composability, and platform independence. After a review of related work on RT programming paradigms, we introduced SL LET, an extension of the LET paradigm that has successfully been exploited for lock-free multicore programming in automotive series designs. We demonstrated how SL LET overcomes the LET limitations by two core concepts, time zones and interconnect tasks, thereby preserving the main benefits of LET.

We showed how to apply SL LET to support programming of two important developments in industrial design, DDS robotics middleware, and TSN networking. We, finally, evaluated the SL LET programming paradigm against the required properties, as well as the induced memory and timing overhead. While the workload overhead that indicates performance loss is negligible and the memory overhead can be widely mitigated by established methods for LET, there is a dependence between available slack and end-to-end latencies that can be used to balance short system response times versus flexibility in composition and change. Importantly, this balance can be safely exploited in design without incurring negative side effects, such as the emergence of race conditions or increased workload. This result was already known for LET. With SL LET, it becomes available for heterogeneous industrial systems integration on a wide range of cyber–physical platforms. ■

REFERENCES

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. 11th IEEE Int. Symp. Object Component-Oriented Real-Time Distrib. Comput. (ISORC)*, May 2008, pp. 363–369.

[2] C. M. Kirsch, "Principles of real-time programming," in *Embedded Software*, A. Sangiovanni-Vincentelli and J. Sifakis, Eds. Berlin, Germany: Springer, 2002, pp. 61–75.

[3] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the LET paradigm," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2018, pp. 240–250.

[4] M. Beckert and R. Ernst, "The ida let machine—An efficient and streamlined open source implementation of the logical execution time paradigm," in *Proc. Int. Workshop New Platforms Future Cars (NPCar at DATE)*, Mar. 2018. [Online]. Available: https://www.date-conference.com/date18/conference/workshop-w03

[5] N. Mansfeld, M. Hamad, M. Becker, A. G. Marin, and S. Haddadin, "Safety map: A unified representation for biomechanics impact data and robot instantaneous dynamic properties," *IEEE Robot. Autom. Lett.*, vol. 3, no. 3, pp. 1880–1887, Jul. 2018.

[6] E. Molina *et al.*, "The autoware framework and requirements for the cognitive digital automation," in *Proc. Work. Conf. Virtual Enterprises*. Cham, Switzerland: Springer, 2017, pp. 107–117.

[7] V. Vyatkin, "The IEC 61499 standard and its semantics," *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 40–48, Dec. 2009.

[8] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Trans. Cyber-Phys. Syst.*, to be published, doi: 10.1145/3381847.

[9] AUTOSAR Consortium. (2018). *Autosar_RS_Timingextensions, Specification of Timing Extensions*. [Online]. Available: https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/MethodologyAndTemplates.zip

[10] C. M. Kirsch and R. Sengupta, *The Evolution of Real-Time Programming* (Computer and Information Science Series), 1st ed. Boca Raton, FL, USA: CRC Press, 2007, pp. 11.1–11.23.

[11] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Berlin, Germany: Springer, 2012, pp. 103–120, doi: 10.1007/978-3-642-24349-3_5.

[12] M. Lohstroh *et al.*, "Reactors: A deterministic model for composable reactive systems," in *Proc. Model-Based Design Cyber Phys. Syst. (CyPhy)*, 2019, pp. 59–85.

[13] N. Halbwachs, "Delay analysis in synchronous programs," in *Computer Aided Verification*, C. Courcoubetis, Ed. Berlin, Germany: Springer, 1993, pp. 333–346.

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

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[16] T. Gautier, P. Le Guernic, and L. Besnard, "Signal: A declarative language for synchronous programming of real-time systems," in *Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Berlin, Germany: Springer, 1987, pp. 257–277.

[17] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Softw. Syst. Model.*, vol. 9, no. 4, pp. 529–565, Sep. 2010.

[18] N. Halbwachs, *Synchronous Programming of Reactive Systems* (Springer International Series in Engineering and Computer Science), vol. 215. Boston, MA, USA: Springer, 1993.

[19] A. Girault, "A survey of automatic distribution method for synchronous programs," in *Proc. Int. Workshop Synchronous Lang., Appl. Programs (SLAP)*, vol. 5, 2005. [Online]. Available: http://www-sop.inria.fr/cma/slap/slap2005.html

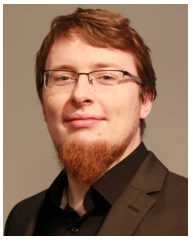
[20] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[21] J. Ouy, "A survey of desynchronization in a polychronous model of computation," *Electron. Notes Theor. Comput. Sci.*, vol. 146, no. 2, pp. 151–167, Jan. 2006.

[22] A. Gamatié and T. Gautier, "The signal synchronous

multiclock approach to the design of distributed embedded systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 5, pp. 641–657, May 2010.
[23] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J. P. Talpin, and S. Tripakis, "A protocol for loosely time-triggered architectures," in *Embedded Software*, A. Sangiovanni-Vincentelli and J. Sifakis, Eds. Berlin, Germany: Springer, 2002, pp. 252–265.
[24] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli, "Heterogeneous reactive systems modeling: Capturing causality and the correctness of loosely time-triggered architectures (LTTA)," in *Proc. 4th ACM Int. Conf. Embedded Softw.*, 2004, pp. 220–229.
[25] A. Benveniste, "Loosely time-triggered architectures for cyber-physical systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2010, pp. 3–8.
[26] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.
[27] T. A. Henzinger, C. M. Kirsch, E. R. B. Marques, and A. Sokolova, "Distributed, modular HTL," in *Proc. 30th IEEE Real-Time Syst. Symp.*, Dec. 2009, pp. 171–180.
[28] W. Pree and J. Templ, "Modeling with the timing definition language (TDL)," in *Model-Driven Development of Reliable Automotive Services*, M. Broy, I. H. Krüger, and M. Meisinger, Eds. Berlin, Germany: Springer, 2008, pp. 133–144.
[29] T. A. Henzinger and C. M. Kirsch, "The embedded machine: Predictable, portable real-time code," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, p. 33, 2007.
[30] S. Resmerita, A. Naderlinger, and S. Lukesch, "Efficient realization of logical execution times in legacy embedded software," in *Proc. 15th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design*, Sep. 2017, pp. 36–45.
[31] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger, "Towards parallelizing legacy embedded control software using the let programming paradigm," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2016, p. 51.
[32] M. Lohstroh and E. A. Lee, "Deterministic actors," in *Proc. Forum Specification Design Lang. (FDL)*, Sep. 2019, pp. 1–8.
[33] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 31–39, Jul. 2005.
[34] C. Farcas and W. Pree, "A deterministic infrastructure for real-time distributed systems," in *Proc. Workshop Oper. Syst. Platforms Embedded Real-Time Appl.*, 2007, pp. 49–57.
[35] A. Naderlinger, J. Pletzer, W. Pree, and J. Templ, "Model-driven development of FlexRay-based systems with the timing definition language (TDL)," in *Proc. 4th Int. Workshop Softw. Eng. Automot. Syst. (SEAS)*, May 2007, p. 6.
[36] *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Standard 1588-2008, Jul. 2008.
[37] M. Di Natale and A. L. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proc. IEEE*, vol. 98, no. 4, pp. 603–620, Apr. 2010.
[38] S. Vitturi, C. Zunino, and T. Sauter, "Industrial communication systems and their future challenges: Next-generation Ethernet, IIoT, and 5G," *Proc. IEEE*, vol. 107, no. 6, pp. 944–961, Jun. 2019.
[39] P. Liggesmeyer and M. Trapp, "Trends in embedded software engineering," *IEEE Softw.*, vol. 26, no. 3, pp. 19–25, May 2009.
[40] L. H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1599–1614, Dec. 2009.
[41] Object Management Group. (2019). *Who is Using DDS?* [Online]. Available: https://www.dds-foundation.org/who-is-using-dds-2/
[42] *Data Distribution Service Object Management Group (OMG)*, document OMG Formal/15-04-10, 2015. [Online]. Available: http://www.omg.org/spec/DDS/1.4/PDF
[43] *DDS Interoperability Wire Protocol*, document OMG Formal/2019-04-03, Object Management Group (OMG), 2019. [Online]. Available: http://www.omg.org/spec/DDSI-RTPS/2.3/PDF
[44] A. Corsaro and D. C. Schmidt, "The data distribution service—The communication middleware fabric for scalable and extensible systems-of-systems," in *System of Systems*, A. V. Gheorghe, Ed. Rijeka, Croatia: IntechOpen, 2012, ch. 2, doi: 10.5772/30322.
[45] H. P. Tijero and J. J. Gutiérrez, "On the schedulability of a data-centric real-time distribution middleware," *Comput. Standards Interfaces*, vol. 34, no. 1, pp. 203–211, Jan. 2012.
[46] H. Pérez and J. J. Gutiérrez, "Modeling the QoS parameters of DDS for event-driven real-time applications," *J. Syst. Softw.*, vol. 104, pp. 126–140, Jun. 2015.
[47] *Time-Sensitive Networking Task Group*. Accessed: Jun. 1, 2020. [Online]. Available: http://www.ieee802.org/1/pages/tsn.html
[48] *Industrial Communication Networks—Fieldbus Specifications—Part 1: Overview and Guidance*, Standard IEC 61158 and IEC 61784 2.0, Apr. 2019.
[49] W. Zimmermann and R. Schmidgall, *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. Wiesbaden, Germany: Springer Fachmedien Wiesbaden, 2014.
[50] *EtherCAT Technology Group*. Accessed: Jan. 15, 2020. [Online]. Available: https://www.ethercat.org/
[51] *PROFINET—The Leading Industrial Ethernet Standard*. Accessed: Jan. 15, 2020. [Online]. Available: https://www.profibus.com/technology/profinet/
[52] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks*, Standard IEEE 802.1Q-2018, Jul. 2018.
[53] *IEEE 802.1AS-Rev—Timing and Synchronization for Time-Sensitive Applications*, IEEE Standard 802.1AS, Draft 8.3, Oct. 2019.
[54] L. Lo Bello and W. Steiner, "A perspective on IEEE time-sensitive networking for industrial communication and automation systems," *Proc. IEEE*, vol. 107, no. 6, pp. 1094–1120, Jun. 2019.
[55] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling real-time communication in ieee 802.1 qbv time sensitive networks," in *Proc. 24th Int. Conf. Real-Time Netw. Syst.*, 2016, pp. 183–192.
[56] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *J. Syst. Archit.*, vol. 80, pp. 104–113, Oct. 2017.
[57] M. Beckert, "Scheduling mechanisms for efficient and safe automotive systems integration," Ph.D. dissertation, Dept. Elect. Eng., Inf. Technol., Phys., TU Braunschweig, Braunschweig, Germany, Oct. 2020.
[58] *SL-LET Prototype Implementation Download, V1.1*. Accessed: Nov. 12, 2019. [Online]. Available: https://www.ida.ing.tu-bs.de/~artifacts
[59] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.
[60] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with Giotto," in *Proc. ACM SIGPLAN Workshop Lang., Compil. Tools for Embedded Syst.*, 2001, pp. 64–72.

ABOUT THE AUTHORS



Kai-Björn Gemlau received the B.S. and M.S. degrees in computer and communication systems engineering from the Technische Universität Braunschweig, Braunschweig, Germany, in 2014 and 2016, respectively, where he is currently working toward the Ph.D. degree.

He is also a Researcher with the Institute of Computer and Network Engineering (IDA), Technische Universität Braunschweig, under the supervision of Prof. R. Ernst, and also a member of the UNICARagil Project. His research interest includes real-time communication for distributed cyber–physical systems, including time-sensitive networking (TSN) networks and network stacks.

Leonie Köhler (née Ahrendts) received the master's degree in electrical engineering from the Technische Universität Braunschweig, Braunschweig, Germany, in 2015, where she is currently working toward the Ph.D. degree at the Institute of Computer and Network Engineering.

Her research interests include the design and analysis of real-time systems with an emphasis on weakly hard real-time systems, fault-tolerant computing systems, and communication networks.

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

After two years at Bell Laboratories, Allentown, PA, USA, he joined the Technische Universität Braunschweig, Braunschweig, Germany, as a Professor of electrical engineering. He chairs the Institute of Computer and Network Engineering (IDA), Technische Universität Braunschweig, covering embedded systems research from computer architecture and real-time systems theory to challenging automotive, aerospace, or smart building applications.

Dr. Ernst is a Fellow of Design, Automation and Test in Europe Conference (DATE). He served as a member of the German Academy of Science and Engineering (acatech). He received the 2014 Achievement Award from the European Design and Automation Association (EDAA). He also served as an Association for Computing Machinery's Special Interest Group on Design Automation (ACM SIGDA) Distinguished Lecturer.