HAL
open science

System-level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software

Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, Sophie Quinton

▶ **To cite this version:**

Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, Sophie Quinton. System-level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software. ACM Transactions on Cyber-Physical Systems, 2021, 5 (2), pp.1-27. 10.1145/3381847 . hal-03125851

HAL Id: hal-03125851

https://inria.hal.science/hal-03125851

Submitted on 1 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software

KAI-BJÖRN GEMLAU, LEONIE KÖHLER, ROLF ERNST, TU Braunschweig, Germany
SOPHIE QUINTON, Inria Grenoble Rhône-Alpes, France

Logical Execution Time (LET) is a timed programming abstraction, which features predictable and composable timing. It has recently gained considerable attention in the automotive industry, where it was successfully applied to master the distribution of software applications on multi-core electronic control units (ECUs). However, the LET abstraction in its conventional form is only valid within the scope of a single component. With the recent introduction of System-Level Logical Execution Time (SL LET), the concept could be transferred to a system-wide scope. This article improves over a first paper on SL LET, by providing matured definitions and an extensive discussion of the concept. It also features a comprehensive evaluation exploring the impacts of SL LET with regard to design, verification, performance and implementability. The evaluation goes far beyond the contexts in which LET was originally applied. Indeed, SL LET allows to address many open challenges in the design and verification of complex embedded hardware/software systems addressing predictability, synchronization, composability, and extensibility. Furthermore, we investigate performance trade-offs and we quantify implementation costs by providing an analysis of the additionally required buffers.

CCS Concepts: • **Computer systems organization → Embedded software**; **Real-time systems**; **Distributed architectures**.

Additional Key Words and Phrases: Embedded software, distributed execution platform, real-time programming models, logical execution time, time determinism, data flow determinism, composability

ACM Reference Format:
Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst and Sophie Quinton. Under minor review. System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software. *ACM Transactions on Cyber-Physical Systems* 0, 0, Article 0 (Under minor review), 27 pages. https://doi.org/x

1 INTRODUCTION

Real-time automotive software applications often implement control algorithms. A team of control engineers provides a functional model, typically programed in Simulink, which serves as a specification of the application. It is challenging for software engineers to translate this platform-agnostic Simulink model into a correct implementation. The process includes function partitioning, function-to-task mapping, signal-to-message mapping, allocation of platform elements to tasks and messages, as well as priority assignment. Moreover, the software engineers have to deal with issues like scheduling, communication, and shared resources. The current ad hoc implementation strategies do not scale well with the increasing complexity of the embedded hardware/software

Authors' addresses: Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, TU Braunschweig, Braunschweig, Germany, gemlau|koehler|ernst@ida.ing.tu-bs.de; Sophie Quinton, Inria Grenoble Rhône-Alpes, Montbonnot, France, sophie.quinton@inria.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© Under minor review Association for Computing Machinery.
XXXX-XXXX/Under minor review/0-ART0 $15.00
https://doi.org/x

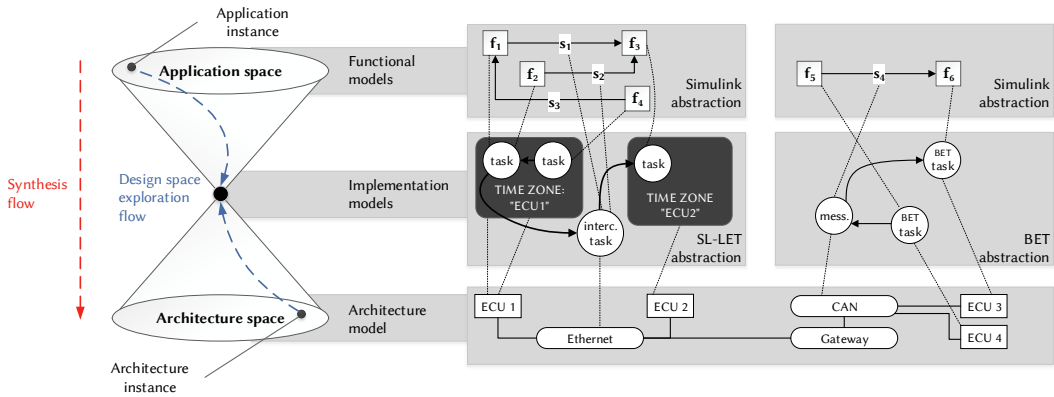


Fig. 1. Design flow with system-level logical execution time (LET) (SL LET). This graphical representation is based on Sangiovanni-Vincentelli et al. [38] and uses SL LET as abstraction for the implementation model.

system, and struggle to meet timing and dependability goals. In particular, the shift from singlecore to a multicore electronic control unit (ECU) has shown that a more formalized and tool-supported design flow is urgently needed in the automotive industry. With the emerging trend to distribute functions over several ECUs in automotive embedded systems (transition from federated to integrated architectures), this consolidated design flow is needed all the more. Sangiovanni-Vincentelli et al. [38] [12] described requirements of a future-oriented and sound embedded system design for automotive applications. Major goals are

• *to decouple the application design from the execution architecture design.* Decoupling the application design and the execution architecture design allows independent development, re-use and portability. The authors of [38] [12] therefore propose a design process which consists in selecting an application instance from the application space (represented by one or more functional models) and selecting an architecture option from the architecture space (represented by an architecture model). An intermediate implementation model should be agnostic of low-level details of the functional model and the architecture model. It is intended as an abstraction which allows to easily derive system properties and evaluate design choices. Moreover, it should preserve the semantics of the functional model in the implementation, including timing properties and data flow. It is an open question what are useful abstractions for the implementation model. The described layered design process is illustrated in Figure 1.

• *to design time-predictable and composable systems.* The real-time behavior of an automotive embedded system must be predictable, i.e. computable, such that time-critical cause-effect chains in the application software can be proven to satisfy their end-to-end deadlines. A key to handling new functionalities and multi-variant products, a very important concern of today's original equipment manufacturers (OEMs), is the composability of automotive embedded systems. Composability ensures the preservation of component properties under composition, and thus significantly eases the important problems of verification and integration.

We agree with [38] [12] that both the described design flow with an intermediate implementation layer and the goal to build time-predictable and composable systems are essential. However, we have identified two further goals which we believe to be a key for the design of complex automotive software systems:

• *to design data-flow deterministic systems.* From the point of view of the functional model, a system consists to a large part of a set of communicating, periodically activated functions. A path

in the graph of communicating functions is called cause-effect chain. Constraints relate to these cause-effect chains: they specify end-to-end deadlines and precedences. A data-flow deterministic system reproduces the precedences specified in the functional model in any simulation run of the implementation model and in any execution run of the implementation.
Timing and data flow are closely related. Time predictability is sufficient to check end-to-end deadlines and to find all possible data flows [2]. However, time predictability is in general not sufficient to create data flow determinism. Data flow determinism is thus an independent requirement.

• *to design programs with platform-independent timing and data flow.* Optimally, the behavior of a program should be independent of the execution architecture with regard to the timing and data flow of cause-effect chains. This allows re-use of application software at a level which goes beyond what is achieved by the layered design approach: Cause-effect chains can then be ported across execution architectures while preserving deadlines and precedences.

One implementation model which embraces all the above named requirements is the logical execution time (LET) programming abstraction [28]. The model represents the system as a set of LET tasks with precedences which are executed on processing elements. It is agnostic of (1) the actual functionality contained in the tasks and (2) the actual architecture of the processing elements. The LET programming abstraction therefore provides the desired abstraction to explore the design space and predict system behavior. Moreover, it is compatible with other implementation models using different abstractions. LET tasks (compared to conventional task models) are assumed to have fixed reading and writing instants at the beginning and end of a static logical execution time. The LET programming uses this time-determinism to achieve time predictability, composability, data flow determinism, and platform independence. At the implementation level, the LET programming abstraction is enforced by hardware/software mechanisms.

Due to all the useful properties and not least because of the data flow determinism, the LET programming abstraction has been recently discovered as a useful approach to master the integration, verification, and implementation of applications on multicore ECUs. Different industrial case studies have been published by OEM and Tier-1 suppliers, which use the LET programming abstraction in this context effectively [42] [21] [37] [31] [32]. The success has even led to an update of the AUTOSAR timing extensions, including now the LET paradigm [10].

However, the LET programming abstraction is limited to a part of the execution architecture, where values written at the end of an LET can be published to potential reader tasks in (nearly) zero time. Unfortunately, the LET-inherent assumption of zero-time communication cannot be fulfilled in a distributed system with remote communication partners. In the context of automated driving where information from many different sensors must be fused to construct the current traffic situation and to take decisions, the direct application of the LET-based strategy must therefore fail.

In this paper, we present the concept of *system-level LET (SL LET)* [14], which deals with the issue of non-zero time communication in distributed computing systems by introducing time zones (where the classical LET paradigm applies) and LET interconnect tasks which link time zones. We show that SL LET preserves the above stated properties of LET. Compared to the seminal paper on SL LET [14], the SL LET concepts are discussed in-depth and an extensive evaluation is added including an actual implementation and case study. We investigate impacts on design and verification, performance trade-offs and quantify implementation costs by providing an analysis of the additionally required buffers.

This paper is organized as follows: We begin with an overview on related work in Section 2. Section 3 details the problems in finding an implementation which satisfies the functional model of an automotive software application. Section 4 presents first the traditional ad hoc implementation

strategy used in automotive industry and the newer LET-based implementation strategy. Then the limitations of the LET programming abstraction are discussed with respect to distributed systems, leading to the concept SL LET which is introduced in Section 5 together with a discussion of important properties of SL LET. Section 6 evaluates the impact of SL LET on design and verification, performance, and implementation cost. The paper concludes with an outlook on open problems.

2 RELATED WORK

This section presents two related programming abstractions, LET and synchrony, which make both assumptions about when a system component reads inputs, when it computes outputs, and how fast outputs are published. The synchronous abstraction is built around the zero-delay concept, while LET abstraction is based on the weaker notion of unit-delay [22]. Both, LET and synchrony, make time explicit in the programming abstraction and the determinism reduces the system state space. Also, the determinism successfully addresses concurrency-related issues like communication and synchronization.

Important differences appear in the design flow. If the synchronous approach is applied, then a functional model is expressed in a synchronous language, e.g. Simulink. This synchronous model is then typically directly compiled to source code. If the LET approach is applied, then the functional model is translated to an intermediate, semantics-preserving LET implementation model. The entities of the LET implementation model are LET tasks with precedence constraints which have a close relation to operating system tasks in a later implementation. LET tasks in the implementation model have predictable, deterministic, composable, platform-independent timing and data flow, and are compatible with different platform-related constraints (legacy tasks, scheduling policies, other programming paradigms, asynchrony etc.). The implementation model is accessible to the software engineers, and is the basis of design space exploration and verification. The existence of this explorable implementation model at a useful abstraction level is in our opinion a strong comparative advantage of LET-based designs over synchrony-based designs in the context of embedded automotive systems. Interestingly, LET and synchrony, have both difficulty to address distributed execution platforms. This is where SL LET comes in as an augmentation of the LET concept.

2.1 Logical execution time systems

Originally introduced as part of the time-triggered programming language Giotto [22] [28], LET abstracts the physical execution time of a task from its logical execution time. It provides deterministic and platform independent communication points at the beginning and the end of a tasks LET. This property was used to separate the scheduling problem from the functionality viewpoint. Consequently it provides a clean interface between the timing model used by the control engineer and that of the software engineer. The concepts of LET have been adapted by hierarchical timing language (HTL) [24] and timing definition language (TDL) [36]. LET-oriented runtime systems include the E machine [23] and Variable-Bandwidth Servers (VBS) [11].

At first the LET concept did not gain much interest in the automotive domain, since the additional costs in terms of memory and timing were still unclear. This situation changed with the introduction of multicore ECUs for safety critical tasks. Due to the massively increased complexity in design and verification of these multicore systems, the benefits of predictability, composability, platform-independence etc. appeared attractive enough – finally outweighing the potential drawbacks of increased latencies and implementation costs. Moreover, the design process needs to be adapted to handle the new complexity [1] and LET has proven to be an efficient contracting interface between the control engineers and the integrators [16]. Implementations from different OEMs

and suppliers [21], [42],[20],[40] paved the way for LET to become integrated into the AUTOSAR standard (version 4.4.0) [10].

Although the LET programming abstraction has become so successful, an important drawback is its limitation to non-distributed systems. In distributed systems, the communication delay becomes non-negligible and the requirement of immediate publishing of data with the elapse of the LET can no longer be fulfilled. To overcome this restriction, one possibility is to hide the finite communication delay within the LET of the message-sending task. This approach has been realized, e.g., by the TDLComm layer published in [18] [17] [34]. Though it leads to transparent distribution in the sense of [18], it entails some serious disadvantages: Firstly, the sender task must have a sufficiently large LET to include both the response time of the task and also the communication delay of the message. Secondly, the communication delay can never be larger than the LET of the sender task LET_{sender} which is a major restriction in the design space. Consider the case of a sender task which transmits a signal by sending samples with the period LET_{sender} such that the Nyquist theorem is satisfied. However, this does not imply that the communication delay to consumer tasks must equal LET_{sender} since the delay can be hidden by pipelining (cf. Figure 5b). Finally, the approach mixes application design and network design which are usually separated fields both from a technical and an organizational perspective. The SL LET concept proposed in this article avoids all three disadvantages.

2.2 Synchronous systems

The behavior of a synchronous system [5] [6] is described by an infinite sequence of atomic reactions, where at each reaction every system component (a) re-computes its outputs based on current inputs and states in zero time, (b) propagates the outputs to other components instantaneously. A reaction thus happens in zero time. As a consequence of this synchrony assumption, synchronous programs are difficult to parallelize due to the need to resolve through fixed-point analysis signal statuses and causality issues. Therefore, a synchronous program is typically compiled to a single-task implementation, which is problematic. In LET programming reading input and writing output is cycle-free so no fixed-point analysis is needed. An exception to the generation of sequential code is the synchronous language PRELUDE [35]. It can formulate a multi-periodic synchronous program, which is then compiled to a multi-task implementation under deadline-monotonic or earliest-deadline-first scheduling while preserving timing constraints and data flow. PRELUDE is not targeted to networked systems.

Synchronous systems are not common in the context of embedded automotive systems. Apart from the common constraint of single-task implementations, an important reason relates to the fact that system design and implementation is a monolithic flow. Also a local design modification may lead to alterations in many system components. Moreover, asynchronous behavior is omnipresent in embedded automotive systems, ranging from interrupts to asynchronous communication infrastructures. The synchronous programming abstraction is not compatible with asynchronous behavior, and can – for instance – handle only synchronous communication media like provided by the time-triggered architectures (TTA) [30].

Finally, we would like to discuss the relation of loosely time-triggered architectures (LTTA) [4], a variation of the synchronous TTA, to SL LET. The concept of LTTA [4] removes the restriction of a globally synchronized clock in comparison to TTA. Instead "Communication by Sampling (CbS)" is used, where data is propagated with independent sampling, communication and writing between non-synchronized clock domains. Due to the possible clock drift between the domains, data losses may occur. The concept of LTTA [4] is related to SL LET, but LTTAs are only defined for time-triggered local scheduling. This is a strong limitation when extending LET to the system level. LTTA uses a register semantics avoiding buffer overflow while back pressure LTTA [39] implements

an "elastic" circuit based on event driven Kahn Graphs, and is therefore not compatible with the LET model. Most importantly, due to unsynchronized clocks, LTTA violates the requirement for cause-effect chain preservation while the proposed back pressure mechanism enforces specific platform protocols which change end-to-end timing and might even be impossible in case of multi-way functional feedbacks. In contrast to that, SL LET [14] [15] does not constraint the underlying scheduling policy and the interconnect tasks can be used to abstract combinations of time triggered and event driven communication. This is possible because SL LET makes the essential assumption of bounded clock drift and jitter.

3 SYSTEM MODEL AND PROBLEM STATEMENT

The need for LET in the automotive industry has arisen from the difficulty to provide a systematic way of building semantics-preserving implementations of the functional models that are developed and validated by control engineers on multicore architectures with partitioned fixed-priority scheduling. In this section, we detail and formalize this problem.

3.1 Functional model

A large part of automotive software mostly consists of control software. Its functional behavior is thus designed by control engineers using abstractions and methods well suited to their discipline [42]. A functional, platform-agnostic model is typically programmed as a block diagram in Simulink that is used to validate the designed system and to generate code. Another modeling option in the context of MATLAB/Simulink is Stateflow, which is a graphical language to specify functionality. It can be translated to Simulink. A block diagram consists of functional blocks and communication links between them. Blocks are activated periodically and communicating blocks must have harmonic periods (one period is a multiple of the other). The Simulink model

- follows a synchronous reactive semantics that is based on the assumption that functional blocks execute in zero (or constant) time; and
- specifies a partial execution order on functional blocks derived from the communication links. This order can be refined to the level of instances of functional blocks by the introduction of unit delays on communication links. This is particularly meaningful for multi-rate systems with undersampling and oversampling.

The completed Simulink model serves as a specification of the automotive functionality to be implemented by software engineers. In practice, a correct implementation of such a model does not satisfy the synchrony hypothesis, but should guarantee that:

- the execution of a functional block always completes before that block is activated again; and
- the execution order on functional blocks is preserved, including possible precedence constraints at the instance level.

For integration and implementation purposes, the Simulink model is mapped to the AUTOSAR software architecture. AUTOSAR describes application software as a set of interacting software components (SW-Cs). An SW-C is internally composed of a set of runnable entities, where a runnable entity (RE) is a sequence of instructions which can be scheduled independently. The Simulink-to-AUTOSAR transformation maps blocks at the upper hierarchical level of a Simulink model onto SW-Cs, while blocks at lower hierarchical levels are mapped onto REs.

An RE is activated on the occurrence of an event. In this case its activation is periodic and the period of an RE is defined as the period of the functional block mapped onto it. Similarly, a deadline constraint is introduced stating that an RE must terminate before its next activation to reflect the corresponding requirement on functional blocks (*implicit deadline*).

SW-Cs can communicate by exchanging signals via a connected pair of sender/receiver ports or request resp. provide services via a connected pair of client/server ports. The communication links between SW-Cs are derived from the Simulink model. The partial execution order on (instances of) REs inherited from the functional blocks mapped on them is translated to a set of explicit precedence constraints. Communication among REs is commonly *implicit*, i.e., an RE copies its set of inputs at the beginning of its execution, executes and finally writes back its set of outputs. In this paper, we assume implicit communication among REs.

We can now formalize the above description as follows.

AUTOSAR-based functional model

Definition 1. A software component SW-C is a graph such that:

- Vertices in the graph represent REs
- Edges represent communication links between (instances of) REs. The expression $\rho \overset{n,m}{\rightarrow} \rho'$ denotes that the mth instance of RE ρ' reads the label value written by the nth instance of RE ρ.

The *AUTOSAR-based functional model* of a system is a set of SW-Cs.

The *requirements*, that an AUTOSAR-based model specifies, are:

- periodic execution of REs with implicit deadlines;
- precedence constraints between REs must be preserved;
- end-to-end deadlines of cause-effect chains in the software component.

Any implementation model of an AUTOSAR-based functional model that satisfies its requirements will be a correct refinement of the system in a synthesis flow.

3.2 Implementation model

The implementation model is an abstraction, which summarizes important properties of the application model and architecture model while abstracting from low-level details. It serves as basis for design space exploration and eventually for integration and synthesis. The concrete entities of the implementation model depend on the chosen abstraction. At this point, we assume that an implementation model will include some abstract representation of the hardware platform as well as tasks to represent software.

Hardware model

Definition 2. A hardware platform is modeled by a *hardware model* $P = \{R, C, M, E\}$ that is an undirected graph made of:

- a set R of computation resources,
- a set C of communication resources,
- a set M of memories, and
- a set E of edges representing the physical connectivity between the resources.

Task

Definition 3. A *task* τ is a piece of code running on a (computation or communication) resource. A task τ_i has a period, a deadline and a priority. It can perform read accesses at the beginning of its execution and write accesses at the end of execution to a set of memory locations called *labels* (read-execute-write semantics).

A task is activated periodically, and an instance of a task τ_i created by the jth activation event is called *job* $\tau_{i,j}$. The execution of tasks concurrently executing on a given resource is arbitrated based on a scheduling policy using static or dynamic priorities.

The mapping of the functional model to the implementation model requires to decide function partitioning, RE-to-task mapping, and possibly signal-to-message mapping. The period and deadline of a task are typically derived from the period and deadline of its REs. In addition, precedence constraints w.r.t. (instances of) REs are transformed into precedence constraints w.r.t. (instances of) tasks. REs mapped onto the same task always execute in the same order so some precedence constraints are trivially verified. The mapping of the of architectural model to the implementation model involves the allocation of platform elements to tasks and messages as well as priority assignment.

We say that an implementation model is schedulable if one can prove that all *task* deadlines as well as *job*-level and *task*-level precedence constraints are satisfied.

3.3 Problem statement

Any implementation model, which is obtained from the Simulink model and the architectural model as explained above and satisfies the originally specified functional requirements, is schedulable and can be synthesized. Obtaining such guarantees in practice is however extremely difficult for software running on multicore platforms and beyond. This process is, in addition, very sensitive to changes in the software and the platform.

The problem addressed in this paper is to create a design flow that (1) decouples application design from execution architecture design, and (2) builds distributed systems that are *time-predictable*, *data-flow deterministic*, *composable* and *platform-independent*. Systems with such properties preserve semantics of the functional model, are significantly easier to dimension and verify (cf. schedulability test, precedence check), and allow plug-and-play of new or modified components.

Time-predictability is a prerequisite for schedulability analysis, which checks whether all specified deadlines can be satisfied.

Time predictability

Definition 4. A system is *time-predictable*, if latencies and jitter of tasks can be computed.

Data flow determinism in sampling task systems is a key enabler for controlling precedence relations.

Data-flow determinism

Definition 5. A system is *data-flow deterministic* if a consumer job $\tau_{i,j}$ always reads a value from the same producer job $\tau_{k,l}$.

The property of composability makes it possible to add or modify a SW-C without altering the behavior of already existing SW-Cs.

Composability

Definition 6. A property is composable if it is preserved through composition.

Finally, platform independence allows to modify the execution platform without loosing the above stated properties.

Platform independence

Definition 7. A system is platform-independent, if time-predictability, data-flow determinism, and composability are preserved through a platform modification.

In the next section, we explain that data-flow determinism was not guaranteed by previous implementation techniques in the automotive industry, which led to the move to LET to cope with the added complexity of multicore platforms.

4 CURRENT USE OF LET IN AUTOMOTIVE

We first describe the situation before the introduction of LET, then explain the current situation with respect to LET.

4.1 Previous solution

In a singlecore implementation, a correct implementation is achieved by carefully designing the schedule which includes the choice of the RE-to-task mapping, the task activation pattern, and the task priorities while assuming a fixed priority scheduling policy. These design techniques can be complemented by introducing rate-transition blocks, that is, adapting the precedence constraints between instances of tasks (this design choice must be inserted back in the Simulink model). Multicore implementations have additionally to deal with concurrency issues, and require the use of synchronization techniques. Nonblocking synchronization techniques are generally preferred since they avoid scheduling anomalies and the danger of deadlocks.

Before the introduction of the LET programming model, no specific mechanisms were used to control data flows. Such implementations followed what we call in the following the *bounded execution time (BET) programming model*, where read and write operations can occur at any time within intervals depending on a task activation, sampling jitter, best-case and worst-case response time. This is illustrated in Figure 2. Such programs are very easy to implement but quite difficult to verify:

First, they are usually not data flow deterministic. Figure 3 shows an example for data flow among a chain of three tasks $\tau_{10ms} \prec \tau_{20ms} \prec \tau_{5ms}$, which are executed on a hardware platform with two cores ($\tau_{10ms} \mapsto r_1$; $\tau_{20ms}, \tau_{5ms} \mapsto r_2$) and a shared memory. On each core, rate-monotonic scheduling is applied. Figure 3a illustrates the difficulty to implement deterministic data flow if the tasks follow the BET programming model: which jobs are involved in a flow, depends on the execution times and thus on the specific execution run.

Second, such programs are not composable. If any BET task in the program is modified, or if a new task/program is added, the entire verification process must be repeated because the timing behavior of all BET tasks in the system may be impacted. Typically, its worst-case response time will change. While data-flow non-determinism could be managed in a singlecore context, it proved too problematic to deal with in a multicore context. This is mainly due to the increased complexity as a result of shared resources like the memory subsystem and independent schedulers on the different cores [7]. Hence the shift towards the LET solution.

4.2 Recent solution: The LET abstraction

The LET programming model is based on deterministic read and write operations. According to the LET programming model, a LET task λ_i is characterized by a deterministic input/output behavior in time. The LET job $\lambda_{i,j}$ is released by an activation event at instant t_r, at which all inputs are read. At a later defined instant $t_w = t_r + LET_i$ all outputs are written. The static time interval between the reading of inputs and writing of outputs, LET_i, is called *logical execution time*. (Note that the LET

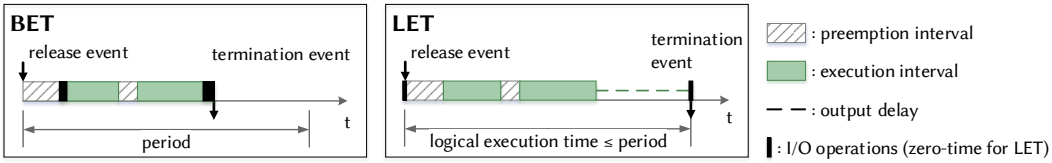
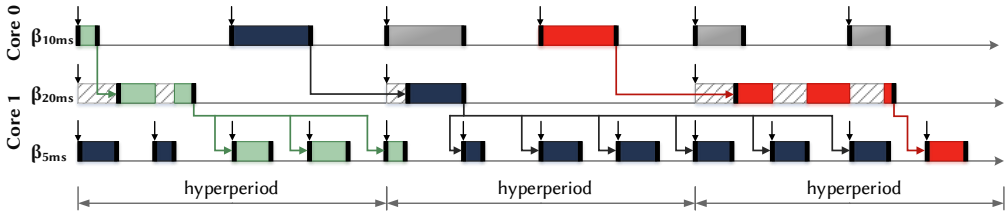
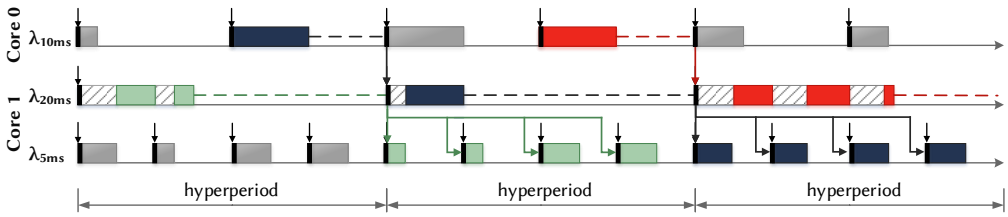


Fig. 2. Timing diagrams for a BET task and a LET task



(a) BET task chain $\beta_{10ms} \prec \beta_{20ms} \prec \beta_{5ms}$: *Tasks have sampling and end-to-end response time jitter. Data flow is varying depending on job execution times.*



(b) LET task chain $\lambda_{10ms} \prec \lambda_{20ms} \prec \lambda_{5ms}$: *Time determinism eliminates sampling jitter and response time jitter while creating static data flow paths.*

Fig. 3. Determinism in time and data flow – comparing the behavior of a task chain with BET resp. LET tasks.

paradigm inherited its name from this central parameter [28]. The context should avoid confusion despite the equivalence in names.) Reading and writing operations are performed in *zero time*. All write values are instantaneously available to all read operations. In contrast to the BET model, the LET programming model is close to the semantics of the functional software model, so it is easy to verify but more complex to implement.

(1) As illustrated in Figure 3b, in any execution run, a consumer job can only read from exactly one producer job due to the fixed input/output phases, which corresponds to the definition of data flow determinism.

(2) In contrast to BET, the LET programming model is composable with respect to the response time of LET tasks. This means that the functionality of a LET task can be extended or an additional LET task can be added without impacting the timing behavior of other LET tasks.

A LET program can be implemented with some effort in the context of AUTOSAR as recently presented in e.g. [3, 8, 32]. A set of software or hardware mechanisms is required to hide variable execution times, to realize (nearly) zero time read/write actions and to deterministically schedule input/output operations. In addition, one must deal with the issue of *label lifetime*, that is, identify for how long label values must be stored for the precedence constraints to be satisfied.

The data flow determinism of the LET programming model, can thus be exploited as an implicit synchronization mechanism as practically demonstrated by [21]. Moreover, since a job reads deterministically from other jobs, the inter-core communication can even be reduced in case of undersampling. In the example, the output produced by the first job of λ_{10ms} in a hyperperiod does not need to be communicated to the second core since it is never used.

4.3 Limitations of the current LET approach

The LET programming model has been successfully applied up to the scope of an automotive multicore ECU [21] and is the focus of intense research in the automotive industry [16]. LET offers strong timing characteristics leading to very desirable, higher-level properties like time determinism, data flow determinism and composability as well as portability and extendability. At the same time the LET programming model is flexible and can be applied with any scheduling algorithm, as long as the read/write timing can be guaranteed. It is thus compatible with static priority preemptive scheduling as applied in the automotive software standards AUTOSAR and OSEK/VDX [9].

While LET has proven to be a good solution to the predictability issue at the ECU level, it is proving however inadequate for the moment at the level of an entire distributed system. The reason is that the LET paradigm requires instantaneous availability of data after their publication. While this is acceptable at the ECU level because of short communication times between cores, communication in a distributed system incurs longer communication delays. In a distributed system, task periods can even be shorter than communication delays, which makes it impossible to "hide" such delays under the logical execution time of such tasks. This was the motivation for the introduction of SL LET.

5 SYSTEM-LEVEL LET

In this section, we formalize system-level LET (SL LET), an extension to the LET paradigm that addresses the issue of implementing LET programs onto distributed platforms.

5.1 Requirements for SL LET

Consider a classical LET program which contains cause-effect chains. Assume that it is partitioned and each partition is executed on a remote element of the execution platform as illustrated in Figure 4. What are the requirements for the communication between the remote partitions of LET programs? What is a potential strategy to satisfy these requirements?

(1) The communication should be *time-predictable*, i.e., a lower and upper bound on the end-to-end latency and the end-to-end jitter of a message can be derived. Following the LET principle, time predictability can be achieved by imposing a time-deterministic communication behavior meaning that the sequence of message transmission times and the sequence of receiving times is identical in any execution run of the system. However, time determinism does *not* imply that time-triggered communication infrastructure is imposed which would not be compatible to event triggered bus and network protocols like CAN or Ethernet that are used today in automotive systems. Furthermore, time determinism should *not* be required at each step of message transmission (for example at each hop in a network), but only at an end-to-end level in form of a *logical end-to-end transmission time* to reduce overhead.

(2) The communication should create *data-flow determinism* between the jobs of the sending and receiving LET tasks. If time-deterministic communication is applied, the same data is available at the sampling instants of the receiving LET task in any possible execution run of the system – given

that the clocks of the remote elements of the execution platform are synchronized with bounded synchronization error (c.f. Section 5.2).

(3) The communication should allow to *compose* new cause-effect chains. In other words, a newly added LET task λ_2 should always be able to subscribe to published data of a remote LET task λ_1. To enable this, the communication service that publishes data of the LET task λ_1 must copy values written by λ_1, such that the remote LET task λ_2 sees them in the same frequency and order as originally produced by λ_1 (necessary and sufficient condition). This can only be achieved by a time-deterministic communication, which transmits a message with the period of λ_1 and delivers it after a logical transmission time. Otherwise the task λ_2 might consume a wrong value in comparison to a non-distributed version of the LET program. This is illustrated in Figure 5b.

Only when a system design is final and all cause-effect chains are known, optimizations may be undertaken like adapting the frequency of the interconnect task to the reading frequency of the consuming tasks in case of under-sampling.

Moreover, the addition of a new communication service should not alter the timing and data flow of already existing tasks and communication services. Composability can be realized by time determinism, where logical execution times and logical transmission times are not affected by new communication services or LET tasks as long as the system is schedulable.

(4) The communication delay should be *platform-independent*. This can again be achieved by time-deterministic communication, which is associated with a logical end-to-end transmission time. As long as a message can be sent and received within the logical transmission time, the externally observable communication delay is independent of the communication medium. At a system-level scale, time determinism (with the consequences of data-flow determinism, platform independence, composability) leads to the very desirable *portability of cause-effect chains* across execution platforms.

In conclusion, we can say that the requirements of time predictability, data-flow determinism, composability and platform independence can without exception be realized by time-deterministic communication which is fully compatible to the LET principles. In some aspects, it is even questionable whether the requirements could be satisfied without recurring to time determinism – in particular with regard to composability and platform independence.

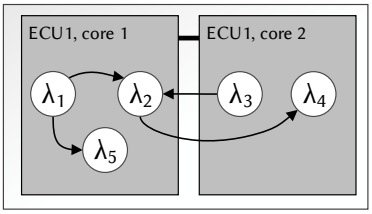
In the following, we formalize this idea and derive concepts which upgrade the LET abstraction to the SL LET abstraction.

5.2 Main concepts for SL LET

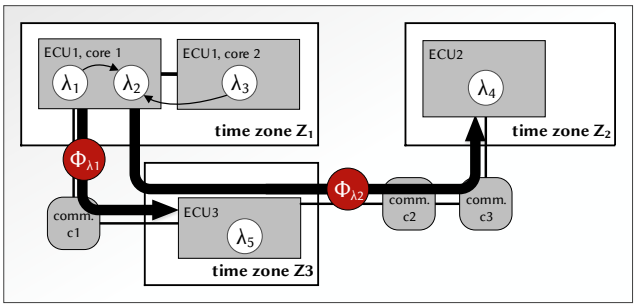
Generally some knowledge on the system architecture and software mapping is available early on in the design exploration process. As a result, one can anticipate which communication will incur some non-negligible delay that cannot be hidden within a regular LET budget. SL LET is based on the concept of *time zone* . Within a time zone Z, communication between LET tasks is considered to be sufficiently fast and predictable to not require any special mechanism on top of the standard LET paradigm. Time zones have each a local time, which are approximates of a global time. The maximum time difference between any two approximates of the global time is bounded from above by a known limited error ϵ.

Synchronization error

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



(a) LET program mapped to elements of an architecture with zero-time communication, cf. [21] [3]



(b) LET program mapped to distributed elements of an architecture with finite-time communication, applying SL LET concepts

Fig. 4. New concepts for applying LET at the system level

In contrast to communication within a time zone, communication between any two time zones is explicit and has a non-negligible delay over the *LET interconnect task*. Immediate and global visibility of data is thus only a requirement within a time zone, while inter-time zone communication deals with delays and clock synchronization.

Time zone decomposition

Definition 9. A *time zone decomposition* of a LET program is a partition of the set of LET tasks. Each element of the partition is called *time zone* Z.

Central to the SL LET paradigm is the concept of LET interconnect tasks, which enable the copying of values between labels in different time zones. Data flows between REs in separate time zones are split into two parts separated by a LET interconnect task, which is a regular LET task that performs the data transport.

Time zone interconnect

Definition 10. A *SL LET interconnect task* (hereafter abbreviated to interconnect task) Φ_{λ_i} is a LET task that copies an output, which is produced by a LET task λ_i and stored in a memory m in a time zone Z_a to a memory m' in a remote time zone Z_b.

Note that we use the notation "Φ_X" in the following to limit the notation complexity. X can be replaced by any annotation and for our examples we use the reference to the producer job. Nevertheless it would be possible to have multiple interconnect tasks per producer or multiple producers per interconnect task.

SL LET programs are thus LET programs enriched with a notion of time zone and LET interconnect tasks. All communication with non-negligible delay happens between time zones and is realized by LET interconnect tasks, which satisfy the requirements stated in section 5.1. In the following, we briefly explain that a SL LET program (and not only the communication or the isolated LET program in a time zone) are time-predictable, time-deterministic, data-flow deterministic, composable, and platform independent.

Time predictability and time determinism. Each LET task belongs to one time zone, where it has static and thus the desired predictable and deterministic timing since it reads and writes labels at defined instants. A LET interconnect task is also time-predictable and time-deterministic (1) in

time zone Z where it reads at defined instants, and (2) in time zone Z' where it produces outputs at defined instants. Since the local time of time zones are approximates of a global time, LET tasks and LET interconnect tasks are also time-deterministic in global time with a bounded error ϵ.

The schedulability of a SL LET program onto a hardware platform can be checked with a classical response time analysis: A LET task or LET interconnect task must not exceed its logical execution time. The SL LET-based model has a good abstraction level for both schedulability tests and design space exploration.

Data flow determinism. The data flow determinism of a SL LET program is a direct consequence of its time-determinism. If both LET tasks and LET interconnect tasks read and write at deterministic instants in time, then the data flow must be deterministic [2]. The advantage of logical end-to-end transmission times over BET transmission times is illustrated in Figure 5. Moreover, if a classical LET program is translated to a SL LET program by adding interconnect tasks, the data flows (and therefore the cause-effect chains) are preserved since task outputs are reproduced in the remote time zone with identical frequency and order.

Composability. The addition of a LET task in a time zone does not alter the timing or data flow of the original LET tasks in that time zone, if all LET tasks can still complete within their logical execution times. The same applies if a LET interconnect task is added to the system to communicate values between two remote time zones Z and Z'. As long as the extra workload for copy and paste operations in each time zone as well as the extra network traffic does not endanger the logical execution times of LET tasks and interconnect tasks, then the timing and data flow is unaffected. In other words, a software or hardware update in an ECU or at the network level has no effect on timing and data flow in the system given that all LETs are still satisfiable.

A LET interconnect task takes care of time-deterministic, order-preserving copying of labels to a remote time zone. It basically provides a data transfer service to which readers in the remote time zone may subscribe. The interconnect task also does not influence how many label values are consumed (e.g. under sampling) and how long label values are kept (label lifetimes), this is decided by the subscribing reader. The separation of concerns creates composability and is in line with the LET programming model and with the publish-subscribe communication commonly used in automotive design (standardized in AUTOSAR). It has also the desirable effect that network design is only responsible for the value transport and is not affected by the choice of label lifetimes.

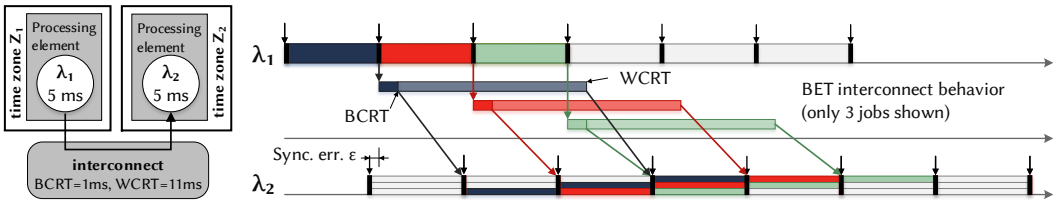
Platform independence. A SL LET program is platform-independent in the sense that as long as the logical execution times of the LET task and the logical transmission times of the LET interconnect tasks are satisfied, a different mapping of LET tasks to processing elements or LET interconnect tasks to communications elements of the execution platform does not alter timing nor data flow.

If re-mapping of a SL LET means that local communications becomes remote communication or vice versa, then LET interconnect tasks appear or disappear in the SL LET program. This does not alter the timing of any task or interconnect task in the system, but it may alter the end-to-end timing (while still keeping the end-to-end deadlines). The deterministic data flow is preserved.

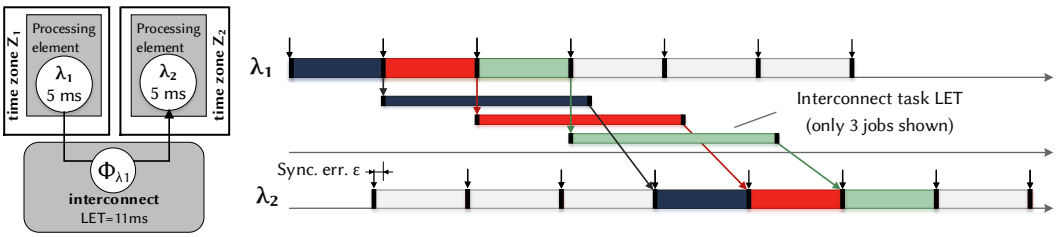
5.3 Implementation of SL LET

As described in Section 5.2, one of the key concepts of SL LET are the *LET interconnect tasks*. An interconnect task Φ is an abstract representation of the communication between two LET tasks in different time zones. In the following, we will describe the benefits of this abstraction and how it can be realized during implementation.

The first step to implement SL LET is to synchronize the local LET schedulers, to provide a common sense of a deadline. This is a fundamental step to ensure that the abstract interconnect task



(a) **Communication with bounded transmission time applied between two LET tasks.** The output sequence of λ_1 as seen by λ_2 varies in each execution run (arrival with jitter, possibly out-of-order). It is thus not identical to the output sequence of λ_1 as seen by λ_2 in a non-distributed version of the LET program, and task λ_2 will consume wrong values.



(b) **Communication with logical end-to-end transmission time applied between two LET tasks.** A unique output sequence of λ_1 is seen by λ_2, which is identical to the output sequence of λ_1 as seen by λ_2 in a non-distributed version of the LET program. This behavior results from the time-deterministic, in-order replay of λ_1-outputs. Altering this replay of outputs in any way violates the property of platform independence and composability of cause-effect chains.

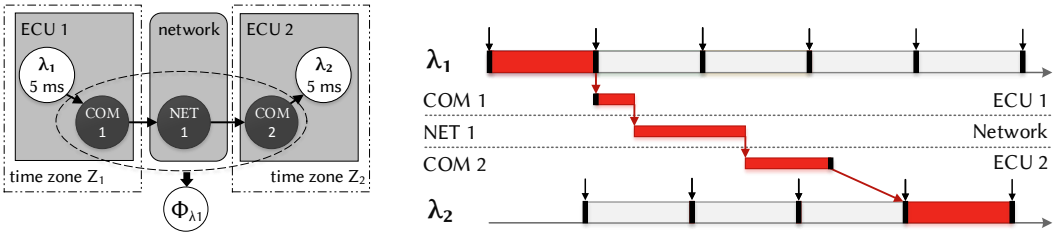
Fig. 5. Importance of using logical end-to-end transmission times for communication

can have LET behavior in both time zones. Without synchronization, the clocks of the distributed ECUs may drift, corrupting the deterministic data flow of LET.

As shown in Figure 6, the interconnect task does not represent a single operating system (OS) task. Instead it includes all the behavior of the end-to-end communication between the remote LET tasks λ_1 and λ_2. This comprises the communication stacks at the sender and the receiver as well as any network communication in-between. For an Ethernet back-end, classical UDP/IP communication stacks might be used, transferring the labels in UDP packets.

Implementing an interconnect task does not constrain the scheduling policy of the underlying network elements. The only requirement is that an upper bound on the end-to-end latency between the communicating time zones is known to dimension the LET of the interconnect task. This compatibility feature allows to create time determinism without having to implement time division multiple access (TDMA) scheduling on the network layer, which is known to carry timing penalties and cost – in particular in the presence of many data paths.

In contrast to a classic LET task, the abstract interconnect task may have a LET larger than the period as shown in the example in Figure 5b. Within a time zone this is not possible, since a job of a task has to finish before the consecutive job starts. In contrast to that, the interconnect task allows multiple jobs to run in parallel, representing the queuing behavior of a network. This implies that multiple instances of a label might be on their way through the network. Moreover, the abstraction allows that label instances take different paths through the network, resulting in out-of-order arrivals at the destination time zone. If label instances are identifiable (e.g., by indexing), then they can be buffered and re-ordered at the destination time zone. A buffered label instance is published



(a) Data flow of the SL LET interconnect task implementation

(b) Timing diagram showing the decomposition of a SL LET interconnect task

Fig. 6. Implementing the abstract SL LET interconnect task

with the elapse of the LET of the associated interconnect task. Our evaluation section discusses indexing strategies and the required size of buffers.

SL LET does not specify what exactly is transferred by an interconnect task. There is no limitation in grouping several LET labels produced in one time zone in one interconnect task. The actual realization of packaging is not restricted given that the input and output behavior is consistent with the interconnect task. This versatility is a result of the abstract nature of interconnect tasks. As an important example, we will evaluate a single sender implementation for multiple destination time zones in Section 6. We will see that though several interconnect tasks are involved, implementation-wise we will only need one sender implementation for all receivers.

The interconnect task may have the same period as the producer task, but the design is not restricted to this choice. In the following we do not focus on further optimizations and assume that labels of one LET task are mapped to one dedicated interconnect task.

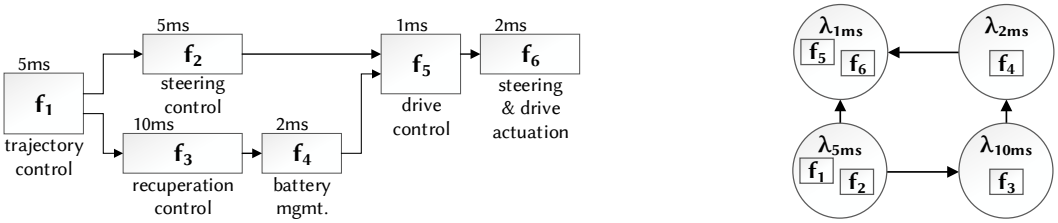
6 EVALUATION OF SYSTEM-LEVEL LET

In this article, we have presented SL LET which is a timed programming abstraction. If SL LET applies in a system, the programmer can rely on a time-deterministic input-output behavior of tasks as well as a time-deterministic inter-task communication with respect to a global time. Timing of components becomes composable with SL LET. As elaborated in Section 3, SL LET serves above all to ease the design and verification of modern complex real-time computing systems. It is intended to replace the conventional programming BET abstraction, whose design and verification processes do not scale well with the complexity of networked multiprocessor systems, multicore processor architectures, and the distribution of software applications.

In the following section, we will evaluate SL LET with an application software example from the automotive powertrain domain. As a starting point for the evaluation, we use the functional representation (simplified MATLAB/Simulink model) given in Figure 7a. It shows a sub-function of the electric powertrain domain, namely to follow a given set trajectory. It also contains execution rates of the different functional blocks as designed by the control engineer.

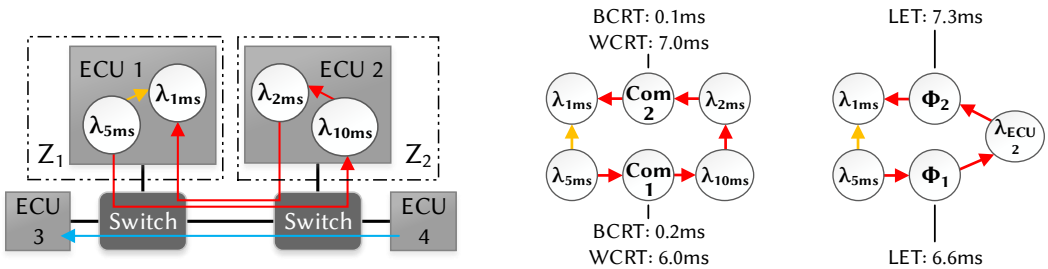
Trajectory following is accomplished by both controlling the steering angle of the wheels (f_2) and adjusting the vehicles speed. For an electric vehicle, recuperation (regenerative breaking) can be used for moderate deceleration, by recharging the battery with the motor (f_3, f_4). For higher deceleration or if the battery cannot handle the additional energy, mechanical brakes are engaged.

Stabilizing the vehicle on a given trajectory therefore is a complex task and the design of the drive control (f_5) relies on synchronized input data from the different paths. However, up to that point the model does not contain any information about a specific hardware platform or who is in charge of developing the sub-functions. Instead, it is agnostic of those details and can be ported to



(a) Functional model (b) Task mapping

Fig. 7. Evaluation example of a powertrain function with a mapping of runnables to tasks



(a) Mapping tasks to ECUs (b) Modeling BET inter-ECU comm. (c) SL LET: Masking the behavior of ECU 2

Fig. 8. Mapping the example cause-effect chains to a distributed hardware

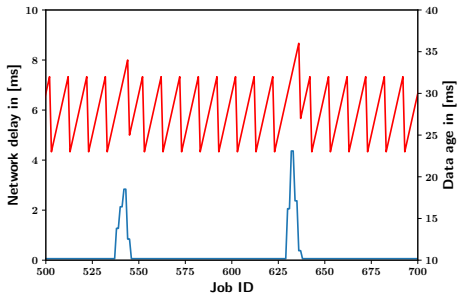
multiple hardware platforms. Based on the Simulink model, program code for the functional blocks is generated (so called *runnables*). Later on, those runnables can be mapped to container tasks that execute with the corresponding or a multiple of the runnables target frequency. An example for a resulting task model is shown in Figure 7b. Again, this does not contain any information about the hardware platform.

In this evaluation section, we use the given example to investigate which challenges in design and verification can be successfully addressed with SL LET. We implemented the example on two distributed ECUs that are connected with commercial Ethernet switches and later extended the setup with our SL LET implementation. More details about this experimental setup are provided in Section 6.3. Then we assess how SL LET impacts performance, i.e, it is asked how efficiently a SL LET-based system performs under a real-time perspective. Finally we evaluate the impact of SL LET on the implementation, i.e., it is analyzed how the SL LET programming abstraction is enforced by hardware/software mechanisms and what is the additional cost.

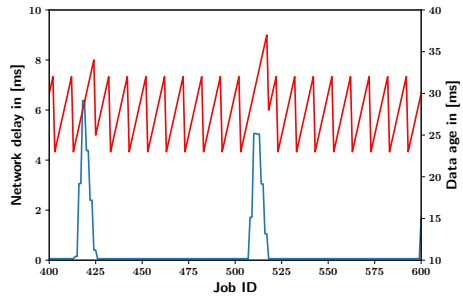
6.1 Impact on Design and Verification

In this section we detail the challenges that appear in the design and verification process of distributed embedded application software with respect to timing, and we discuss how SL LET can help to address them.

The design of modern embedded software is highly collaborative. A major challenge is how to consider this *division of labour* in the timing specification of the software. A timing specification should always include the deadlines of the time-critical cause-effect chains which are part of the software.



(a) Non-deterministic data age for the red cause effect chain



(b) Modified data age due to updated network traffic (larger network delay)

Fig. 9. Without SL LET: Measured data ages at the end of the red cause-effect chain in Figure 8a

Using the example from Figure 7, the different functions might be mapped to a given hardware platform as shown in Figure 8a. In this scenario, ECU 1 is in charge for all drive and steering control operations, typically called a powertrain controller. The recuperation functionality is part of ECU 2, an energy management ECU, which is closely related to the battery system. Both are connected to an Ethernet backbone which also transmits traffic from other ECUs (blue arrow). Assume ECU 1 and ECU 2 are developed by different suppliers that need to protect their intellectual property (IP). Although both teams can use a local LET scheduler, the Ethernet communication will be modeled as BET tasks in the red cause-effect chain (Figure 8b).

The measured data in Figure 9a show how a bursty network delay (blue) leads to an non-deterministic data age for the red cause-effect chain from Figure 8b. Due to the oversampling in the given cause effect chain, the ideal data age that is observed by a job of the 1ms task is a sawtooth waveform. Whenever network delay varies, this directly affects the measured data age, although both local LET schedulers are synchronized in this setup.

Of course, the measured data age from Figure 9a can be fed back to the control engineer's Simulink model by adding delay blocks e.g. with the measured average or worst-case network delay. Nevertheless the introduced jitter cannot be represented in the model and both teams can not rely on a deterministic data age of their input data. Whenever an update changes the communication jitter, the timing behavior of the BET communication task changes again as shown in Figure 9b. Both teams then need to redo the tests and iterate again to the control engineer's Simulink model, where the timing change possibly requires control parameter adaptation. The result is a complicated design process as we know it today.

SL LET can be used beneficially here by introducing a vertical as well as a horizontal division of labour between the stakeholders. First of all, the jitter-free data delivery enables an implementation that actually represents the Simulink model. Since the implementation reflects the model, function verification can be done based on the model, or a model based prototype, without expensive function testing at the level of the final implementation. All that is needed is a test or analysis that the implementation does not violate the LET timing. The BET communication tasks from Figure 8b are replaced by the SL LET interconnect tasks, whose properties act as a contracting interface between the different stakeholders (horizontal division of labour). Especially when IP must be protected, the least amount of information can be used for contracting, while still enabling predictable timing behavior. An example is given in Figure 8c, where the internals of ECU 2 are hidden with an abstract representation. The only information that is required for the developer team of ECU 1 is the input

and output behavior with the interconnect tasks Φ_1 and Φ_2 as well as the timing budget for the processing on ECU 2.

Furthermore, SL LET allows a top-down approach, where the deadline of a cause-effect chain can be decomposed into independent timing budgets (logical execution times) for chain segments, thereby dividing the design into separate subtasks. SL LET exploits the common separation of ECU / microcontroller design and network design, specifying LETs for computation and communication segments of the cause-effect chain. A computation chain segment represents a set of dependent LET tasks within a specific time zone, while a communication segment corresponds to an LET interconnect task transferring labels between time zones. The LET of a chain segment can always be refined by the responsible working group to sub-LETs in a hierarchical manner. Related to the example of Figure 8c, the LET of the interconnect task can be defined larger than the worst case response time (WCRT) of the transmission, enabling robustness to future modifications. Changes within the network domain are not visible to the other stakeholders, as long as the WCRT is kept smaller than the defined LET. As soon as the interconnect LET needs to be modified, this has to be propagated to all stakeholders and can be reflected in the Simulink model. Therefore the overall iterations between controller design and integration are reduced to a minimum. This can be further exploited when introducing LETs and sub-LETs with different scopes: While LETs are visible to all stakeholders, sub-LETs are only visible to the individual working group and allow changes without notifying others. In contrast to systems following the BET paradigm, the timing of chain segments is fully composable which eases integration and verification.

A timing specification for a chain segment, which can be interpreted as a component, should *preserve the semantics of the component-specific modeling approaches and tools*. While the BET insufficiently preserves semantics of control models, SL LET is sufficiently close to the synchronous reactive semantics of MATLAB/Simulink, which assumes a constant execution time of functional blocks.

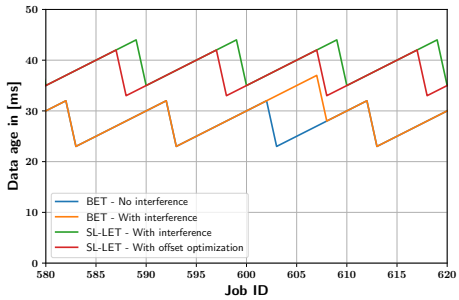
Moreover, it is desirable to relieve the designer from the burden of adapting the component design to *changes of the execution platform*. These changes may be caused by new ECU architectures, new network architectures, updates of the basic software or a different application mapping. SL LET conveniently hides these execution platform-related effects by a set of hardware/software mechanisms. Because the timing behavior of the component remains unaffected, the component designer may even be unaware of changes. This also leaves flexibility to platform architects, basic software developers and system integrators.

The *plug-and-play of subsystems* in an existing design is an important issue in current automotive system design due to multi-variant products as well as software upgrades and updates. The plug-and-play capability requires a low-effort (re)verification and is therefore closely related to a clearly deterministic and composable timing behavior of components and the communication between components, which SL LET provides.

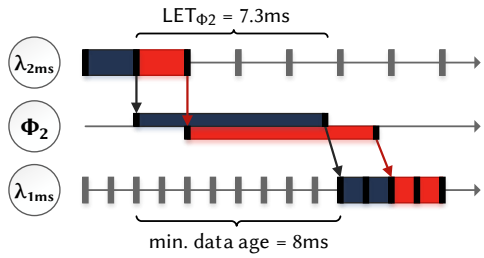
6.2 Impact on Performance

One of the most frequent arguments against LET, also applicable to SL LET, is the increased latency. In contrast to the BET programming abstraction where the worst-case latency is a rare situation, (SL) LET makes the worst-case latency to be the common case if no optimization is applied. In this section, we will discuss the impacts of SL LET on the performance and argue why latency is important but needs to be balanced against jitter and robustness.

Modern control software (e.g. from the automotive powertrain domain) consists often of distributed cause-effect chains with multiple involved ECUs. These distributed control applications require often low to zero jitter to maintain control quality. However, inter-ECU communication under BET adds a significant jitter to the cause-effect chain. Furthermore, this jitter impacts other



(a) Measured data age at the output of the red cause-effect chain



(b) Comparing interconnect task LET with data age

Fig. 10. Deterministic data age with SL LET

cause-effect chains which share the same communication medium. SL LET can help to eliminate this jitter at the cost of increased latencies. An increased latency due to SL LET affects the controller's dead time, but constant latency has also advantages as it can enable advanced control algorithms like model predictive control [16].

Based on the previously given example, Figure 10 shows the measured data age at the output of the red cause effect chain from Figure 8 (including inter-ECU communication). As already mentioned, the data age is not constant since it is a multi rate system, including different levels of over- and under-sampling. Nevertheless, the data age is deterministic for a given job id in a hyperperiod which in this case leads to a sawtooth waveform, exactly as in the underlying synchronous model. The exact type of waveform for the hyperperiod depends on the setup of the cause-effect chain. According to Feiertag et al. [19], the upper and lower bound for the data age can be derived by the "last-to-first" and "last-to-last" semantic for such a multirate system.

Figure 10a shows, that a BET inter-ECU communication *might* also result in a deterministic data age, as long as there is no interference on the network (blue line). But as soon as (in this case bursty) network traffic leads to a higher transmission delay, the data age is not deterministic anymore (orange line). For the concrete example, the maximum data age increases up to 15% (42ms instead of 37ms). Taking a look at a single job makes the situation even worse. While job ID 605 is supposed to read 25ms old data, it now gets input data that is 35ms old, which is an error of 40%. Any job based optimization of the control algorithm therefore is disturbed.

By applying SL LET in this experimental setup, the data age increases by 52% (green line) but the determinism is preserved even in case of bounded interference. As shown in Figure 8c, this is achieved by defining the interconnect tasks LET (7.3ms) slightly larger than the WCRT for that network transmission (7ms). The overhead is also required to cover the synchronization error between the local LET schedulers.

This raises the question how the data age is affected by the chosen interconnect tasks LET. Figure 10b illustrates the correlation between the interconnect task Φ_2 and the observed data age at the input of λ_{1ms}. Although the interconnect tasks LET is 7.3ms long, the first job that reads the blue value observes a data age overhead of 8ms due to SL LET. Consequently the interconnect tasks LET might be increased to 8ms without affecting the data age, but introducing additional robustness to increased network jitter. As a result, robustness and data age are correlated with a discrete (stepwise) function, which enables future optimizations according to the semantics proposed by Feiertag et al. [19].

Several optimizations have been proposed to reduce the latency in the context of LET, and they are also applicable to SL LET. One common method, that we applied to our experimental setup, is to delay the read of a task by an offset such that it is synchronized with the write of the task from which it reads its inputs [29]. This optimization is also applicable to SL LET, since the interconnect tasks behave exactly like LET in the source and destination time zones. Moreover the interconnect tasks have an arbitrary LET (based on their WCRT). As a result, it is likely that their write accesses have a constant offset to the readers activation. Related to the given example, this can be used beneficially for the under-sampling case between Φ_1 and λ_{10ms}. Since the interconnect task has a period of 5ms and a LET of 6.6ms, it will always provide its data 3.4ms before the next job of λ_{10ms} is activated. By delaying the activation of λ_{10ms} with a 7ms offset, the data age of the whole cause-effect chain could be reduced about 2ms as shown in Figure 10a (red line). It is important, that the effect of such an offset depends on the over- and under-sampling in the latter part of the cause-effect chain (e.g. in the example the end-to-end reduction amounts 2ms, although λ_{10ms} observes 3ms reduction). Nevertheless SL LET enables this optimization for a distributed cause-effect chain due to the explicit knowledge of read and write times.

In addition to that, the trend goes towards higher flexibility during the products life-cycle. Due to a more disruptive design process, updates or even new software that was not developed during the first roll-out has to be applied to a working system. Consequently, the question about the robustness of cause-effect chains towards future updates is raised. SL LET offers a good insight into extensibility and robustness properties of the system. The robustness margin of a SL LET task or SL LET interconnect task is given by the difference between its LET and its actual worst-case response time in the current configuration. As long as the robustness margin of any task involved in a cause-effect chain is not exceeded by a modification of the system, the real-time properties of this cause-effect chain will not be affected. Moreover, this robustness margin can be used to identify bottlenecks that are critical for future changes. With BET, robustness properties are less obvious and the timing behavior of the entire software system would to be re-analyzed in case of a modification.

In summary, latency has been considered as a primary performance factor but with modern distributed software application other performance aspects like the reduction of jitter and the robustness towards modifications have become of growing importance. SL LET offers a trade-off where an increase in latencies is balanced against flexible, jitter-free and robust computation and communication. Moreover SL LET is able to provide explicit knowledge for applying optimization techniques on the whole cause-effect chain which was not possible before.

6.3 Impact on Implementation

In this section, we identify the additional implementation costs which are incurred by SL LET. We evaluated the concept of SL LET with an implementation that can be downloaded here [25]. As shown in Figure 11, the hardware platform used for the evaluation consists of two Xilinx ZCU102 Boards [41] that communicate with UDP/IP over two commercial Ethernet switches. Both ECUs run μC/OS-II [33] as an operating system and the lwIP communication stack [13]. Furthermore, the LET implementation from [3] is adapted to be synchronized between both ECUs with the precision time protocol (PTP). Two Linux hosts are used to generate interference traffic on the communication path between λ_{2ms} and λ_{1ms}.

Compared to a system with local LET and BET-based inter-ECU communication, SL LET introduces the interconnect task Φ. As explained above, the interconnect task is an abstract concept, which does not correspond to a single operating system task. Instead, it comprises the communication stacks at the sender side and the receiver side as well as the network in between. All of these

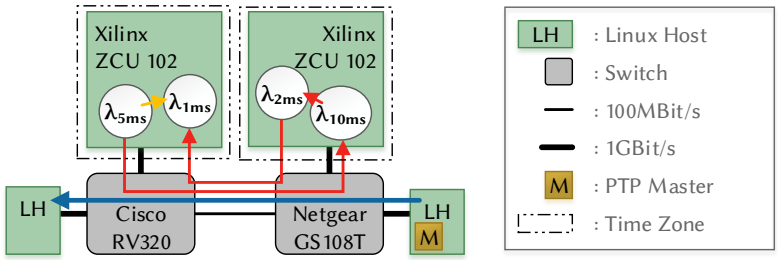


Fig. 11. Hardware setup for evaluation

parts have already been necessary for BET-based inter-ECU communication. In addition to that, SL LET requires

- *time synchronization* between the ECUs which determine the time zones in this case, and
- *time-deterministic packet delivery.*

In the following, we discuss both aspects.

Time synchronization is already part of modern distributed systems and the *precision time protocol (PTP)* is typically used (e.g. defined in IEEE 1588v2 [26]). Besides an initial synchronization between the PTP master and the PTP slave, periodic re-synchronization is required to mitigate clock drift. Insofar, we can expect that the system implementation already provides a global time with controllable deviation.

It is important to mention that the local LET schedule needs to be synchronized to that global time. Common LET implementations rely on a general purpose timer that is used to generate activation interrupt requests (IRQs) according to the LET schedule [3]. On the other hand, the timer that is used for the PTP time synchronization resides in the Ethernet hardware. If both timers need to be synchronized, the additional error can be reflected by the SL LET synchronization error ϵ

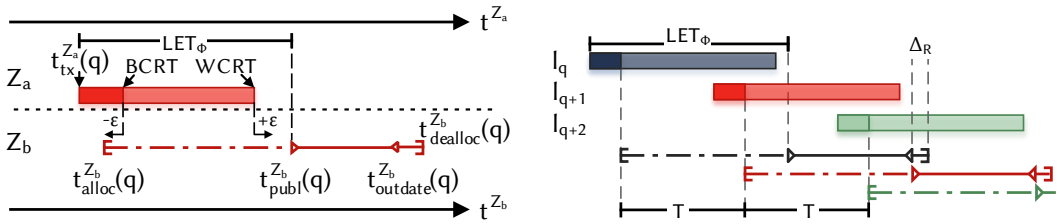
$$\epsilon = \epsilon_{ptp} + \epsilon_{ptp-let} \tag{1}$$

Modern hardware platforms like the Infineon Aurix [27] or the Xilinx Ultrascale+ MPSoC [41] include Ethernet hardware that is capable of generating IRQs based on the PTP timer. This minimizes the synchronization error and reduces implementation overhead. Due to the fast initial synchronization, short startup-times are possible which is required in the automotive domain. For our evaluation, we used a Linux host as PTP master, resulting in a sufficient synchronization accuracy of 500ns. In summary, time synchronization can be assumed as already given and is not an overhead caused by SL LET.

In the implementation, time-deterministic packet delivery relies on *sequence numbering* at the sender and *buffering and reordering* at the receiver. The additional sequence number for a packet is an insignificant overhead compared to the typical payload, where the latter is in the order of tens to hundreds of bytes. Moreover, packets need to be buffered until their data is to be published for the consumer according to SL LET. Since there can be a large variance in packet traversal times, also reordering of packets might be required at the receiver side.

Buffers represent the most important part of SL LET implementation cost. Consequently, we derive the number of required buffer entries at a receiver task under the assumption of harmonic task periods.

We begin with deriving the life time $LT(q)$ of a buffer entry B_q, which stores the label instance l_q. The life time $LT(q)$ corresponds to the delay between the allocation time $t_{alloc}(q)$ and the



(a) Transmission of label l_q from time zone Z_a to Z_b (b) Arrival of 3 consecutive labels in time-zone Z_2

Fig. 12. SL LET buffering at the receiver side

de-allocation time $t_{dealloc}(q)$

$$LT(q) = t_{dealloc}(q) - t_{alloc}(q). \tag{2}$$

As shown in Figure 12a, the instant $t_{alloc}(q)$ should be the earliest point in time when l_q can arrive in the time zone of the receiver task because an empty buffer entry must then be available.

If the interconnect task sends with the period T, the resulting transmission times with respect to the the sender timezone Z_a are then given by

$$t_{tx}^{Z_a}(q) = q \cdot T. \tag{3}$$

Due to the variable network delay, the data will arrive at an arbitrary point in time

$$t_{rx}^{Z_a}(q) \geq t_{tx}^{Z_a}(q). \tag{4}$$

The earliest point in time when a buffer needs to be available for reception is bounded by minimizing $t_{rx}^{Z_a}(q)$ to the best case response time (BCRT) of the network

$$t_{alloc}^{Z_a}(q) = t_{tx}^{Z_a}(q) + BCRT. \tag{5}$$

Given the bounded synchronization error (Definition 8), a lower bound for the allocation time in time zone Z_b is

$$t_{alloc}^{Z_b}(q) = t_{alloc}^{Z_a}(q) - \epsilon = t_{tx}^{Z_a}(q) + BCRT - \epsilon. \tag{6}$$

The instant $t_{dealloc}(q)$ is the latest point in time when l_q is outdated and no reader is accessing the buffer entry. Let us call $t_{outdate}^{Z_b}(q)$ the latest point in time when label l_q is outdated in time zone Z_b. Since there is no possibility to read in zero-time in a real-world implementation, the buffer entry has to be preserved a bit longer, namely for the duration of the longest possible read-phase of any consumer of this label Δ_R

$$t_{dealloc}^{Z_b}(q) = t_{outdate}^{Z_b}(q) + \Delta_R. \tag{7}$$

According to the LET paradigm, the label instance is outdated as soon as the consecutive instance becomes published (ref. triangle symbol in Figure 12a).

$$t_{outdate}^{Z_b}(q) = t_{publ}^{Z_b}(q+1) \tag{8}$$

Since the publication time $t_{publ}^{Z_b}(q)$ depends on the LET of the interconnect task, a safe lower bound for the LET can be given as

$$\begin{aligned} t_{publ}^{Z_b}(q) = t_{tx}^{Z_a}(q) + LET_\Phi \geq t_{tx}^{Z_a}(q) + WCRT + \epsilon \\ LET_\Phi \geq WCRT + \epsilon \end{aligned} \tag{9}$$

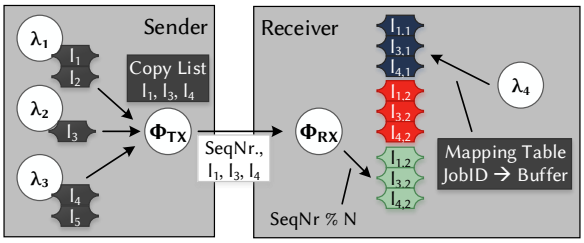


Fig. 13. Example SL LET implementation [25]

As a result, the life time $LT(q)$ of a buffer entry can be derived as

$$\begin{aligned} LT(q) &= t_{dealloc}(q) - t_{alloc}(q) \\ &= T + LET_{\Phi} + \Delta_R - BCRT + \epsilon. \end{aligned} \tag{10}$$

As one can see, the life time is independent of the instance q, hence we will call it LT in the following. During the life time LT, new buffer entries are allocated with the sender period T as shown in Figure 12b. Therefore, the required number of buffer entries N for the given (de)allocation rule is

$$N = \left\lceil \frac{LT}{T} \right\rceil = 1 + \left\lceil \frac{LET_{\Phi} + \Delta_R - BCRT + \epsilon}{T} \right\rceil. \tag{11}$$

In conclusion, buffering is required to hide the network jitter but the amount of memory can be safely bounded as described above. Please, note that the synchronization error epsilon is only determined by the clock synchronization between sender and receiver time zones. Intermediate time zones which are passed in between do not affect the error ("fly over", see [15]).

Figure 13 shows the proof-of-concept implementation [25] that was implemented based on the assumptions made above. Note that, besides the constant memory overhead, SL LET can be implemented with a constant runtime overhead. The sender part of the SL LET interconnect task Φ_{TX} is activated according to LET in the sender time zone and gathers labels that need to be transferred. Those labels are transmitted in conjunction with a sequence number. Besides this sequence number (which is a negligible overhead) the sender side is identical for the BET and the SL LET case, leading to a measured runtime of around 26µs for the transmission path. The receiver task Φ_{RX} is in charge of storing the data in the right buffer entry. This write operation uses the sequence number modulo the number of buffers ($SeqNr\%N$) to identify the right storage. As a result, the buffer holds the correct sequence of values, masking of out-of-order arrivals. Consequently, the runtime overhead is negligible leading to a measured receive delay (processing in the communication stack) of around 40µs for the BET as well as the SL LET case. The buffering is the only important memory overhead since with SL LET there are N buffer entries for each label available while with BET there is only one entry per label. On the other hand, a consumer task λ_4 that wants to read a label is redirected with a specific mapping table to the corresponding buffer based on its job id. The mapping exploits the fact that the consumer job $\lambda_{4,i}$ always reads from the same producer job $\Phi_{TX,j}$ within a hyper-period. This example shows again that SL LET can be implemented with extremely low runtime overhead.

In summary, SL LET can be implemented with a constant memory and nearly zero runtime overhead. Techniques for time synchronization and sequence numbering are already part of modern distributed systems and can be re-used without further modifications.

7 CONCLUSION

The LET concept has been introduced in industrial practice to master the design challenges of safety critical multicore systems by introducing time determinism, data-flow determinism and composability. Nevertheless, modern cause-effect chains are distributed over multiple ECUs, including BET communication in-between. SL LET can be used here, preserving the determinism that is otherwise lost.

In this paper, we took a detailed look on SL LET, by formulating the underlying requirements and elaborating the core concepts. We provide a complete implementation to evaluate the buffering and timing overhead. An implementation concept for the abstract interconnect task was provided, that does not constrain the scheduling paradigm of the intermediate network. The proposed implementation has a constant memory and runtime overhead and is therefore applicable to embedded real-time systems with limited resources. Moreover, the practical use of SL LET in the automotive design and verification process is shown. SL LET can be used here as a contracting scheme and fits well the separation of concerns between ECU design and network design. Therefore SL LET is able to close the gap in the current development process, preserving the determinism from the control model down to the implementation of a distributed cause effect chain.

Like LET, SL LET is currently defined for periodic tasks. This is no fundamental limitation of SL LET and shall be addressed in future work in combination with an experimental implementation of SL LET.

ACKNOWLEDGMENT

We thank Hermann von Hasseln and Julien Hennig from Daimler AG for valuable discussions and feedback from the industrial practice.

REFERENCES

[1] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. 2014. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.* 13, 4 (2014), 82:1–82:37. https://doi.org/10.1145/2560033

[2] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. 2017. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture* 80 (2017), 104–113.

[3] Matthias Beckert and Rolf Ernst. 2018. The IDA LET Machine - An efficient and streamlined open source implementation of the Logical Execution Time Paradigm. In *International Workshop on New Platforms for Future Cars (NPCar at DATE 2018)*.

[4] Albert Benveniste. 2010. Loosely Time-Triggered Architectures for Cyber-Physical Systems. In *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*. 3–8. https://doi.org/10.1109/DATE.2010.5457246

[5] Albert Benveniste and Gérard Berry. 2002. The synchronous approach to reactive and real-time systems. *Readings in hardware/software co-design* (2002), 147–159.

[6] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[7] Alessandro Biondi and Marco Di Natale. 2018. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 240–250.

[8] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. [n.d.]. Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores. ([n. d.]).

[9] AUTOSAR consortium. 2018. AUTOSAR standards. https://www.autosar.org/standards/.

[10] AUTOSAR consortium. 2018. AUTOSAR_RS_TimingExtensions, Specification of timing extensions. https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/MethodologyAndTemplates.zip.

[11] Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. 2009. Programmable temporal isolation through variable-bandwidth servers. In *IEEE Fourth International Symposium on Industrial Embedded Systems, SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8-10, 2009*. 171–180. https://doi.org/10.1109/SIES.2009.5196213

[12] Marco Di Natale and Alberto Luigi Sangiovanni-Vincentelli. 2010. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proc. IEEE* 98, 4 (2010), 603–620.
[13] Adam Dunkels. Accessed: 2019-05-01. lwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/
[14] Rolf Ernst, Leonie Ahrendts, and Kai-Björn Gemlau. 2018. System Level LET: Mastering Cause-Effect Chains in Distributed Systems. In *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 4084–4089.
[15] Rolf Ernst, Leonie Ahrendts, Kai-Björn Gemlau, Sophie Quinton, Hermann von Hasseln, and Julien Hennig. 2018. *System Level LET with Application to Automotive Design (Technical Memorandum)*. Technical Report. Institute of Computer and Network Engineering, TU Braunschweig.
[16] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. 2018. Dagstuhl Seminar 18092 – The Logical Execution Time Paradigm: New Perspectives for Multicore Systems. http://drops.dagstuhl.de/opus/volltexte/2018/9293/pdf/dagrep_v008_i002_p122_18092.pdf.
[17] Claudiu Farcas and Wolfgang Pree. 2007. A deterministic infrastructure for real-time distributed systems. In *OSPERT 2007 Workshop on Operating Systems Platforms for Embedded Real-Time applications*.
[18] Emilia Farcas, Claudiu Farcas, Wolfgang Pree, and Josef Templ. 2005. Transparent distribution of real-time components based on logical execution time. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 31–39.
[19] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. 2009. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society.
[20] Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. 2014. Formal Analysis of Timing Effects on Closed-Loop Properties of Control Software. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*. 53–62. https://doi.org/10.1109/RTSS.2014.28
[21] Julien Hennig, Hermann von Hasseln, Hassan Mohammad, Stefan Resmerita, Stefan Lukesch, and Andreas Naderlinger. 2016. Towards parallelizing legacy embedded control software using the LET programming paradigm. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*. IEEE, 1–1.
[22] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. 2003. Giotto: a time-triggered language for embedded programming. *Proc. IEEE* 91, 1 (2003), 84–99.
[23] Thomas A. Henzinger and Christoph M. Kirsch. 2007. The embedded machine: Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.* 29, 6 (2007), 33. https://doi.org/10.1145/1286821.1286824
[24] Thomas A. Henzinger, Christoph M. Kirsch, Eduardo R. B. Marques, and Ana Sokolova. 2009. Distributed, Modular HTL. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*. 171–180. https://doi.org/10.1109/RTSS.2009.9
[25] IDA. Accessed: 2019-05-01. SL-LET implementation download. https://www.ida.ing.tu-bs.de/~artifacts
[26] IEEE. Accessed: 2019-05-01. IEEE 1588-2008- IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. https://standards.ieee.org/standard/1588-2008.html
[27] Infineon. Accessed: 2019-05-01. *32-bit TriCore™ Aurix™ – TC2xx*. https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/
[28] Christoph M. Kirsch and Ana Sokolova. 2012. The Logical Execution Time Paradigm. In *Advances in Real-Time Systems*. 103–120. https://doi.org/10.1007/978-3-642-24349-3_5
[29] Tomasz Kloda, Bruno D'Ausbourg, and Luca Santinelli. 2014. Towards a more flexible timing definition language. In *12th International Workshop Quantitative Aspects of Programming Languages and Systems-at ETAPS 2014*.
[30] Hermann Kopetz. 2003. Time-triggered real-time computing. *Annual Reviews in Control* 27, 1 (2003), 3–13. https://doi.org/10.1016/S1367-5788(03)00002-6
[31] Stefan Kuntz. 2018. Multicore and Logical Execution Time, Industry's Perspective. http://materials.dagstuhl.de/index.php?semnr=18092. *Dagstuhl Seminar 18092 – The Logical Execution Time Paradigm: New Perspectives for Multicore Systems* (2018).
[32] Ralph Mader. 2018. Implementation of Logical Execution Time – in an AUTOSAR based embedded automotive multi-core application. http://materials.dagstuhl.de/index.php?semnr=18092. *Dagstuhl Seminar 18092 – The Logical Execution Time Paradigm: New Perspectives for Multicore Systems* (2018).
[33] Micrium. Accessed: 2019-05-01. Real-Time Kernels: µC/OS-II and µC/OS-III. https://www.micrium.com/rtos/kernels/
[34] Andreas Naderlinger, Johannes Pletzer, Wolfgang Pree, and Josef Templ. 2007. Model-driven development of flexray-based systems with the timing definition language (TDL). In *Fourth International Workshop on Software Engineering for Automotive Systems (SEAS'07)*. IEEE, 6–6.
[35] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. 2011. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems* 21, 3 (2011), 307–338.
[36] Wolfgang Pree and Josef Templ. 2006. Modeling with the Timing Definition Language (TDL). In *Model-Driven Development of Reliable Automotive Services, Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Revised Selected Papers*. 133–144. https://doi.org/10.1007/978-3-540-70930-5_9

[37] Stefan Resmerita, Andreas Naderlinger, Manuel Huber, Kenneth Butts, and Wolfgang Pree. 2015. Applying real-time programming to legacy embedded control software. In *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*. IEEE, 1–8.

[38] Alberto Sangiovanni-Vincentelli and Marco Di Natale. 2007. Embedded system design for automotive applications. *Computer* 40, 10 (2007).

[39] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto L. Sangiovanni-Vincentelli, Paul Caspi, and Marco Di Natale. 2008. Implementing Synchronous Models on Loosely Time Triggered Architectures. *IEEE Trans. Computers* 57, 10 (2008), 1300–1314. https://doi.org/10.1109/TC.2008.81

[40] Franz Walkembach. 2016. Model-driven Development for Safety-critical Software Components. White paper. http://events.windriver.com/wrcd01/wrcm/2016/08/WP-model-driven-development-for-safety-critical-software-components.pdf

[41] Xilinx. Accessed: 2019-05-01. Xilinx Zynq Ultrascale+ Device - Technical Reference manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

[42] Dirk Ziegenbein and Arne Hamann. 2015. Timing-aware control software design for automotive systems. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. 56:1–56:6. https://doi.org/10.1145/2744769.2747947