System Level LET with Application to Automotive Design

Rolf Ernst, Leonie Ahrendts, Kai-Björn Gemlau
Institute of Computer and Network Engineering
TU Braunschweig
Braunschweig, Germany
ernst|ahrendts|gemlau@ida.ing.tu-bs.de

Sophie Quinton
Inria Grenoble Rhône-Alpes
Montbonnot, France
sophie.quinton@inria.fr

Hermann von Hasseln, Julien Hennig
Daimler AG
Stuttgart, Germany
hermann.v.hasseln@daimler.com,
julien.hennig@daimler.com

***Abstract*—The logical execution time (LET) programming model has been applied in the automotive industry to master multicore programming of large task systems with complex dependencies. Recent developments in electric powertrains and autonomous vehicle functions raise parallel programming from the multicore level to the vehicle level where the requirements for LET application do not hold any more. This paper introduces System Level LET (SL LET), an extension of LET with relaxed synchronization requirements. While related extensions have been proposed for specific scheduling and communication models before, SL LET can be used with a variety of scheduling algorithms and communication semantics. Furthermore, it can be applied to systems with combinations of LET and other programming models. Yet, SL LET allows end-to-end timing guarantees and preserves essential LET properties required for automotive systems. For illustration, we apply the model to an electric vehicle use case.**

***Index Terms*—Embedded software, distributed real-time systems, logical execution time, time determinism, data flow determinism, composability**

I. Introduction

Multicore implementations brought new challenges to automotive software development. Established methods of placing executables called runnables into container tasks combined with static priority scheduling failed to scale to concurrent execution on multiple cores leading to dependency and data consistency problems even in purely periodic application systems. The main source of these problems is the non-buffered memory coupled task communication, which cannot easily be replaced in the typical automotive multi-rate task sets.

To master the multicore design process, there is a growing interest in programming models with defined execution and communication timing. One possible approach is the logical execution time (LET) programming model [1]. LET adopts a periodic task model and introduces time into programming by fixing the times when data are read and written to memory. Fixed reading and writing times constrain scheduling but can be used to enforce execution sequences and mutual exclusive data access in a periodic system. Different approaches exploiting such fixed execution times have been proposed by industry, e.g. communication at the implicit task deadlines at the end of a period [2] or, more fine grain, within a period [3], [4]. Fixing communication times is generally seen as an opportunity to reach a more deterministic interface in the automotive supplier-OEM development chain and ease later changes and reuse. A small loss in implementation efficiency, in particular a larger cause effect chain latency, does not outweigh the gain in predictability [5] [6]. There is currently an initiative by OEMs and suppliers to include LET timing in the automotive software architecture standard, AUTOSAR, as an extension to the existing timing module [7].

Besides multicore design, there is a second trend in automotive systems, application distribution over several electronic control units (ECUs) in a vehicle. Examples are advanced driver assistance system (ADAS) all the way to automated driving, where many high-end sensors (cameras, radar, lidar, ...) contribute to sensor fusion, or future electric vehicles where multiple electric drives must be tightly coupled for flexible steering and motion. With the positive experience in using LET to master the multicore design process, there is a strong incentive to extend the use of LET to distributed systems. Unfortunately, the larger delay times between ECUs do not allow applying the LET as it is, at least not at the needed level of efficiency. This paper proposes an extension to LET which relaxes the LET synchronicity assumptions thereby adhering to the requirements of the automotive design process and is compatible to the existing LET programming model.

This paper is organized as follows: After this introduction, section II reviews the LET programming model and its use in the automotive industry. Section III provides an assessment of the LET limitations when extending the scope to distributed systems. Section IV introduces an extension of LET, which we call *System Level LET (SL LET)* with the two fundamental constructs of communication tasks and time zones. It shows the compatibility to the existing LET semantics and investigates the impact of clock differences on communication and end-to-end timing properties. On that basis, it derives requirements to time zone synchronization such that the requirements from the use in the automotive design process, as elaborated in section II, are met. Section V discusses the implementation of time zones and the use of communication tasks in the design process. Section VI applies SL LET to an automotive use case. Section VII surveys related work and discusses applicability of the basic constructs to

Technical report, August 2018

related programming models. Section VIII concludes and gives an outlook on open issues in LET based design.

II. The LET concept and its current use in automotive software

A. Functional Model

Automotive software mostly consists of control software, with the consequence that its functional behavior is designed by control engineers using abstractions and methods well suited to their discipline. A functional, platform-agnostic model is typically programmed as a block diagram in Simulink that is used to validate the designed system. The completed Simulink model serves as a specification of the automotive functionality to be implemented by software engineers. In the implementation process, the Simulink model has to be mapped to the AUTOSAR software architecture.

AUTOSAR describes application software as a set of interacting software components (SW-Cs). SW-Cs can exchange signals via a connected pair of sender/receiver ports or request resp. provide services via a connected pair of client/server ports. Signals and services are transported over the virtual functional bus (VFB), which is an abstract communication infrastructure. An SW-C is internally composed of a set of runnable entitys (REs), where an RE is a sequence of instructions which can be scheduled independently. An RE is activated on the occurrence of an event, mostly in a periodic manner. 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 periodic activation and implicit communication among REs. The Simulink-to-AUTOSAR mapping is such that the blocks at the upper hierarchical level of a Simulink model represent then SW-Cs, while blocks at lower hierarchical levels map to REs.

The implementation process, which should preserve the specified system behavior in time, is intricate since (1) the Simulink model follows a synchronous reactive semantics with the idealistic assumption of zero time execution or a constant execution time of functional blocks (i.e. REs) and (2) the Simulink model specifies a partial execution order of functional blocks (i.e. REs). The software engineer has to relax the unimplementable requirement of zero time execution of an RE. Typically, a deadline constraint is introduced stating that an RE must terminate before its next activation. The partial execution order of REs is translated by the software engineer to a set of explicit precedence constraints. An RE ρ precedes an RE ρ' ($\rho \rightarrow \rho'$), if ρ completes execution before ρ' is activated. Precedence constraints exist not only among REs but also, in particular for multi-rate systems with undersampling and oversampling, among instances of REs. We say that the nth instance of RE ρ precedes the mth instance of RE ρ' ($\rho \overset{n,m}{\rightarrow} \rho'$), if the nth instance of RE ρ completes execution before the mth instance of RE ρ' is activated.

B. Implementation Model

The AUTOSAR system model comprises at the implementation stage the hardware platform with ECUs, gateways, data buses and networks etc. as well as the associated software on each processing element. The software on an ECU has three layers: the application layer including the SW-Cs, the runtime environment (RTE), and the basic software (BSW). The RTE implements the ECU-related functions of the VFB, while the BSW modules provide communication, memory and system services. Facing this complex hardware/software architecture, appropriate scheduling mechanisms have to be selected which enforce deadlines and precedence constraints in the implementation but avoid at the same time deadlocks and scheduling anomalies. Scheduling decisions include the mapping from REs to tasks, the static execution order of REs within each task, the assignment of scheduling priorites to tasks, and the control of rate transition between tasks. Note that with the mapping of REs to tasks, the deadline constraint for an RE translates to a deadline constraint for the corresponding container task. Precedence constraints w.r.t. REs are transformed into precedence constraints w.r.t. tasks while precedence constraints w.r.t. to instances of REs are transformed into precedence constraints w.r.t. to jobs. Thus, now *task* deadlines as well as *job*-level and *task*-level precedence constraints have to be satisfied.

For software on multicore platforms and beyond, the refinement process from the functional model to the implementation has almost come to a dead end due to the complex scheduling decisions to be taken. This situation has been motivating the automotive industry to alter the execution behavior of the hardware/software system, transitioning from the classical bounded execution time (BET) programming model to the LET programming model. In the following, we define both the BET and LET programming model. Some of the definitions are shared by both programming models; we present them first. Then we give definitions in which the models differ. Later in section II-C, we will see the advantages of the LET programming model over the BET programming model.

Definition 1 (Computation and communication resource, service): A *computation resource* r_i provides processing service, while a *communication resource* c_i provides communication service. The provided processing or communication service is measured in time units. Service of a given computation or communication resource is arbitrated between service-demanding entities according to a given scheduling policy.

Definition 2 (Memory): A *memory* m_i stores program code, private data and/or shared data.

Definition 3 (Label): A *label* l_k is a data structure which contains a *value* v at a given point in time.

Definition 4 (Hardware platform): A *hardware platform* P consists of a set of computation resources R, a set of communication resources C, a set of memories M and a set of edges E representing the physical connectivity between the resources. A hardware platform $P = \{R, C, M, E\}$ can thus be modeled as an undirected graph.

Definition 5 (Task): A *task* τ_i is an entity which consumes service when executed on a service-providing resource. A *computation task* consumes processing service on a computation resource r, while a *communication task* consumes communication service on a communication resource c. A task τ_i has a static or dynamic priority depending on the scheduling policy applied at the resource on which it is executed. A task is executed periodically, an instance of a task τ_i created by the jth activation event is called *job* $\tau_{i,j}$. The response time behavior of task τ_i is constrained by a relative deadline d_i, which corresponds to the period (*implicit deadline*). Each task τ_i may perform read accesses at the beginning of execution and write accesses at the end of execution to a set of labels L_i (*read-execute-write semantics*).

The BET and LET programming models, illustrated in Figure 1, differ in the assumptions they make on the implementation of a task.

*a) **Bounded Execution Time Programming Model**:* The BET programming model is close to the physical reality of program execution on a real computing platform.

Definition 6 (BET task): A BET task β_i is a task, where the service demand of a job of task β_i ranges between its best case execution time (BCET) C_i^- and the worst case execution time (WCET) C_i^+. The jth instance of an BET task is the *BET job* denoted as $\beta_{i,j}$.

*b) **Logical Execution Time Programming Model**:* The LET programing model is close to the semantics of the functional software model. It can be implemented with some effort in the context of AUTOSAR as recently presented [6], [8], [9]. 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.

Definition 7 (LET task): A LET task λ_i is a task, which is characterized by its deterministic input/output behavior in time. The jth instance of a LET task is the *LET job* denoted as $\lambda_{i,j}$. 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*. Reading and writing operations are performed in *zero time*. All write values are instantaneously available to all read operations.

C. Benefits of the LET Programming Model For Automotive Software

The LET programming model receives increasing attention in the automotive industry [10] since it is an efficient means to find a correct implementation of the functional model. In practice, the LET programming model has been successfully applied up to the scope of an automotive multicore ECU [3]. The LET programming model features determinism both in execution time and input/output behavior. These basic timing characteristics lead to very desirable, higher-level properties like time determinism, data flow determinism and composability as well as portability and extendability. At the same time

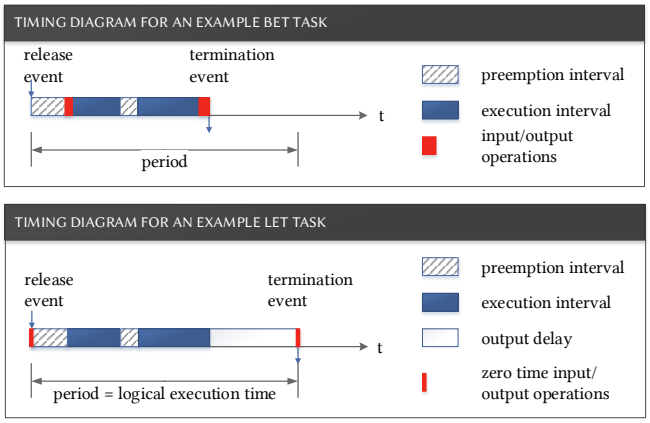


Fig. 1: Timing diagrams for a BET task and an LET task

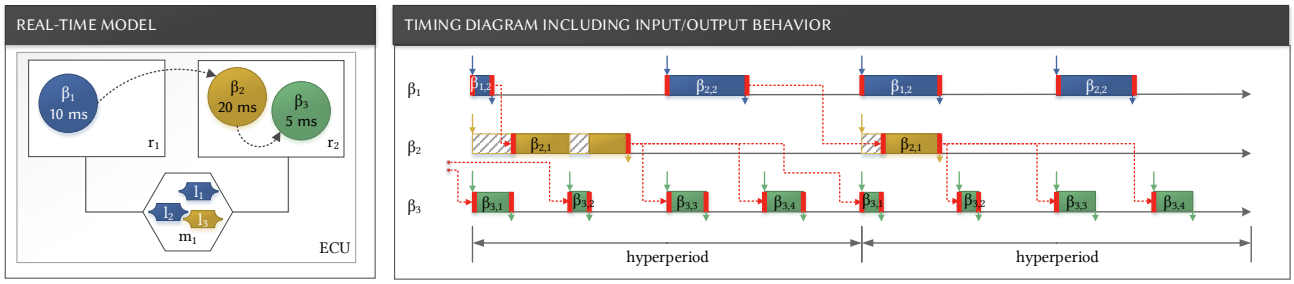
the LET programming model is flexible and can be applied with any scheduling algorithm, as long as the read/write timing can be guaranteed. It is thus compatible with static priority preemptive scheduling as applied in the automotive software standards AUTOSAR and OSEK/VDX [11].

1) Time determinism: The functional model of an automotive application software as described in section II-A makes idealistic assumptions which ease the design process for the control engineer [2]. These idealistic assumptions comprise (1) zero time reading of inputs at the release event, (2) zero or constant execution time of computation, and (3) zero time writing and propagation of outputs. This leads to zero sampling jitter, where sampling jitter is the variable delay between the release of a job and the action of reading inputs. Another direct consequence is zero response time jitter, where response time jitter refers to the variable delay between the release of a job and its termination.

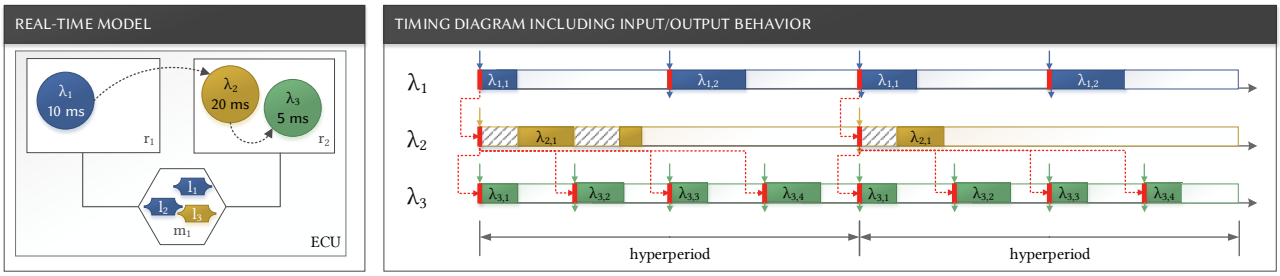
If tasks follow the BET programming model, the assumptions (1)-(3) are not valid in the implemented system, and either these effects must be integrated in a post-implementation simulation model or functional deviations must be accepted. The LET programming model actually imitates the behavior of the functional model, since it enforces the assumptions (1)-(3) and realizes the desirable property of time determinism [12].

Definition 8 (Time determinism): As a response to a given sequence of inputs, a *time-deterministic task* produces a sequence of outputs which has the same timing in any execution run.

2) Data flow determinism: When implementing a functional model, then the data flow – formally characterized by precedence constraints – must preserved in the implementation. In a singlecore implementation, this 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. Multicore implementations have additionally to deal with concurrency issues, and require the use of synchronization techniques. Nonblocking synchro-



(a) BET task chain: *Tasks have sampling and end-to-end response time jitter. Data flow is varying depending on job execution times.*



(b) LET task chain: *Time determinism eliminates sampling jitter and response time jitter while creating static data flow paths.*

Fig. 2: Determinism in time and data flow – comparing the behavior of a task chain implemented with BET resp. LET tasks.

nization techniques are generally preferred since they avoid scheduling anomalies and the danger of deadlocks.

Figure 2 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. Subfigure 2a 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. This is different if the task follow the LET programming model, as illustrated in Subfigure 2b. In any execution run, a consumer job can only read from exactly one producer job due to the fixed input/output phases. We call this property data flow determinism.

Definition 9 (Data flow determinism): A pair of communicating tasks $\tau_k \prec \tau_i$ is *data flow-deterministic*, if each consumer job $\tau_{i,j}$ reads a value always from the same producer job $\tau_{k,l}$ in any execution run.

The data flow determinism of the LET programming model, can thus be exploited as an implicit synchronization mechanism as practically demonstrated by [3]. 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.

3) Composability: Compositionality means that the properties of a composite can be derived from the properties of its components, while composability implies that the properties of individual components are preserved on component composition [13]. An important consequence of the BET programming model is that the worst case response time of BET tasks is not composable but compositional. Thus if any BET task is modified or added, the timing behavior of all BET tasks in the system is changed so that the entire verification process has to be re-performed. In contrast, 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.

III. LET for Distributed Systems – An Assessment

With the LET programming model being so convenient at the ECU level, it is desirable to transfer the concept to the entire distributed system profiting from composability, time determinism and data flow determinism in this extended scope. Once introduced at the system level, these properties can develop their full potential in system design and verification. In fact, there are no reasons against applying the conventional LET programming model to a distributed system if the problem is considered from an implementation-agnostic perspective. Once, however, the assumptions of the LET programming model are compared to what is actually implementable in a distributed system, it becomes clear that the concept does not scale. There is no reality which fits the model: (1) On the one hand, the LET programming model requires zero time reading/writing of data together with immediate, global visibility of data. On the other hand, communications times are non-negligible in distributed systems. (2) A related challenge are large differences in communication times in distributed systems. Some (local) values of an input data set might be

already overwritten, while other (remote) input values are still in the process of being communicated. This can lead to considerable inconsistencies in the data age of inputs, corrupting the desired functional behavior. (3) Furthermore, we have to deal with local times in distributed systems and thus with the issue of synchronicity. In the following we detail each of the named challenges and sketch a possible solution.

A. Visibility of Outputs

At the ECU level, zero time writing/reading and instant visibility of data is achieved, e.g., by implementing double buffers which are stored in scratch pads [8]. This solution is no longer viable for distributed systems, where the long communication delay cannot be hidden by a skillful implementation. Thus communication delays among distant senders and receivers must be made explicit. To preserve time determinism and data flow determinism, the sending event must be periodically scheduled and the communication delay between the sending event and the receiving event must be static. The static communication delay is also required to preserve the desirable composability of response times and communication times under the LET programming model.

B. Label Lifetime

A label is a data structure containing a value that can be overwritten, cf. Definition 3. Overwriting implicitly limits the lifetime of the written value to the next write operation, like in typical programming languages. Due to the short communication times in memory coupled multicore architectures, labels can be read before they are overwritten at the component level. Figure 2b gives an example for a multi-rate application with LET tasks. In multi-rate applications, where a writer can produce more values than a reader consumes, the application either assumes a last-is-best semantic by only reading the latest value or it uses some intermediate task that reduces the written samples to a single value possibly preprocessing the value to reduce the output signal bandwidth (e.g. median filter). So, even in multi-rate systems, the overwriting issue is not of practical importance with multicore architectures. The situation changes in distributed systems. Consider Figure 3a, where in a single-ECU implementation task λ_2 reads from tasks λ_1, λ_3, and where the input values (here: $\lambda_{1,4}$, $\lambda_{3,1}$) originate from the same hyperperiod. In a distributed implementation shown in Figure 3b, the communication of a value of λ_1 (here: $\lambda_{1,4}$) has a considerable delay such that the matching value of λ_3 (here: $\lambda_{3,1}$) is overwritten before $\lambda_{1,4}$ arrives. Consequently a temporal inconsistency of input data age occurs, so that input data for task λ_2 comes from different hyperperiods. This temporal inconsistency might not be acceptable. We should, therefore, consider not to assign every output of a producer task τ_i to one and the same label using an overwriting mechanism, but to assign the value of each producer job $\tau_{i,j}$ to an individual label (*single assignment label*). Then e.g. the value $\lambda_{3,1}$ is buffered, so that the temporal inconsistency can be eliminated. This gives rise to another challenge: that we have the property of a single assignment variable, as often used in parallel compiler optimization. In periodic systems, we would need an unbounded number of value labels because we cannot know which of those labels will (later) be used in a composed system with unknown communication delay times. Hence, when introducing single assignment labels, we must explicitly bound their lifetimes. Note that single assignment labels can also be of interest, if values can arrive out-of-order due to alternative communication paths.

C. Synchronicity

The LET model assumes a global time base providing identical timing for all cores and memories. Like communication times, possible clock phase variations add to timing uncertainty and require additional time between write and read operations. Clock drift leads to unbounded uncertainty and is therefore not compatible to LET. In summary, worst case communication time and clock phase variation add up defining the minimum time between read and write operations on any LET label. In distributed systems these times will dominate task execution time and, hence, dominate the design of a distributed system. Moreover, communication depends on the composition and integration of system functions and the type and load of the communication network. In conclusion, the definition of LET labels will be highly influenced by the network design countering the idea of composability thereby letting a separated local multicore LET implementation and component reuse appear impractical. In the following section, we will introduce an extension of LET that avoids this dilemma.

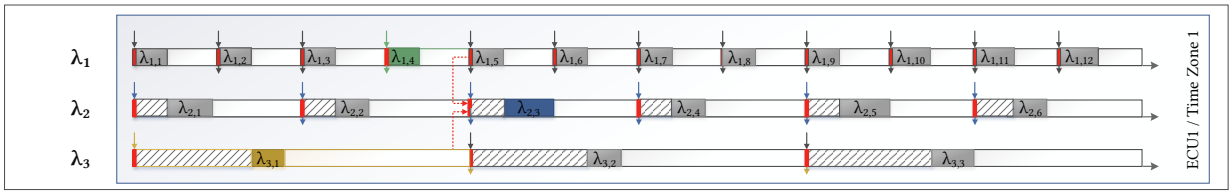
IV. System Level LET

We propose System Level LET (SL LET) to solve the scalability issues discussed in the previous section. SL LET is a comprehensive paradigm, which integrates the LET programming model in a set of more general concepts. In the following, we will detail these concepts and show that SL LET preserves the properties of composability, time determinism and data flow determinism.

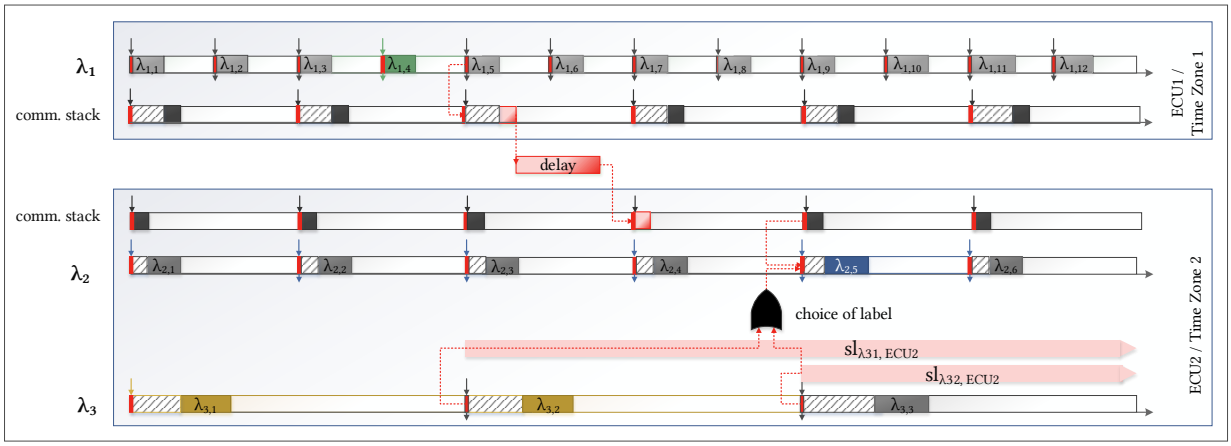
A. Concepts

We introduce the notion of time zones and time zone interconnects as illustrated in Figure 4. Time zones have each a local time, which are approximates of a global time. Within a time zone, the original LET programming model applies. Communication between any two distant time zones is explicit and has a non-negligible delay over the time zone interconnect. 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.

Definition 10 (Time zones): A time zone $Z_i = \{R_{Z_i}, C_{Z_i}, M_{Z_i}, E_{Z_i}\}$ is a subgraph of the hardware platform P. Any two times zones Z_i, Z_j of the hardware platform P are disjoint: $\forall i, j: \ Z_i \cap Z_j = \emptyset$. Within a time zone Z_i, the LET programming model is valid. In particular read and write operations to labels, which are stored in memories in the time zone Z_i, are performable in zero time. Each time zone Z_i



(a) Single-ECU implementation: *Tasks λ_1, λ_3 provide inputs to task λ_2. All inputs for λ_2 are from the same hyperperiod.*



(b) Distributed implementation: *Due to different communication times, the inputs for λ_2 are no longer from the same hyperperiod – unless data is appropriately buffered by single assignment labels.*

Fig. 3: Introducing single assignment labels to eliminate inconsistent data ages

has a local time $t_{Z_i}(e)$, which associates every event e with a timestamp.

Definition 11 (Time zone interconnect): A time zone interconnect $I_i = \{C_{I_i}, E_{I_i}\}$ is a subgraph of the hardware platform P which consists only of linked communication resources. A time zone interconnect I_i is not part of any time zone ($\forall i, j: \ I_i \cap Z_j = \emptyset$), but it connects one or more time zones. Any two times zone interconnects I_i, I_j of the hardware platform P are disjoint: $\forall i, j: \ I_i \cap I_j = \emptyset$.

Definition 12 (Platform composition): The union of all time zones $Z = \bigcup Z_i$ contains all computation resources and all memories ($R \subset Z$, $M \subset Z$). The set of edges $E_{Z,I}$ contains all edges which connect time zone elements with time zone interconnect elements. The union of all times zones $Z = \bigcup Z_i$, the union of time zone interconnects $I = \bigcup I_i$ and the set of edges $E_{Z,I}$ constitute the hardware platform $P = Z \cup I \cup E_{Z,I}$.

Definition 13 (Synchronization): A synchronization mechanism creates a global time basis among all time zones. The local time of a time zone is a local approximate of the global time basis. The maximum time difference between any two approximates of the global time is bounded from above by a known limited error ϵ

$$\forall i, j: \ \epsilon = \max |t_{Z_i}(e) - t_{Z_j}(e)|.$$

As an important consequence, SL LET does not allow arbitrary clock drift, but limits the permissible local phase and clock jitter w.r.t. the global clock. This requirement is essential to keep the fundamental synchronity properties of LET and is a major differentiator to concepts allowing communication between subsystems with non synchronized clocks.

Communication between time zones means copying values between distant labels. Label lifetimes were identified to be an issue in distributed systems, when the copying delay is non-negligible. We require that a label lifetime is a purely time-zone dependent decision to ensure composability. To realize the requirement, we refine now our notion of labels such that (1) instances of the same label may reside in different time zones, and (2) the value of each producer job $\tau_{i,j}$ may be assigned to an individual label with an explicit lifetime.

Definition 14 (Label, instance of a label): A *label* l_{τ_i} is a data structure, which is written repeatedly by the producer task τ_i under a defined overwriting semantics. An *instance* of the label l_{τ_i} in the memory m is denoted as $l_{\tau_i,m}$.

Definition 15 (Single assignment label, instance of a single assignment label): A *single assignment label* $sl_{\tau_{i,j}}$ is a data structure, which is written once by the producer job $\tau_{i,j}$. An *instance* of the single assignment label $sl_{\tau_{i,j}}$ in memory m is denoted as $sl_{\tau_{i,j},m}$. The time of origin $origin(sl_{\tau_{i,j},m})$ indicates when the output of the producer job $\tau_{i,j}$ is written to $sl_{\tau_{i,j},m}$. The label lifetime $life(sl_{\tau_{i,j},m})$ defines how long the output of the producer job $\tau_{i,j}$ is stored in $sl_{\tau_{i,j},m}$.

Central to the SL LET paradigm is the concept the LET interconnect task, which enables the copying of values between labels in different time zones. While the LET interconnect task takes care of time-deterministic copying, it is agnostic of label lifetimes. The LET interconnect task reads a label to

be published in another time zone, while the receiving time zone can extend the lifetime to the extent necessary in that time zone. This separation of concerns 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.

Definition 16 (LET interconnect task): A *LET interconnect task* ϕ_{λ_i} is a generalized LET communication task. It copies an output, which is produced by a LET task λ_i and stored in a memory $m \in Z_a$, to a remote memory $m' \in Z_b$. The LET interconnect task λ_i consumes service on a composed resource, namely on all communication and processing resources involved in the copying process between the times zones Z_a, Z_b. The jth instance of a LET interconnect task is the *LET interconnect job* denoted as $\phi_{\lambda_{i,j}}$. This job copies the content of the label $sl_{\lambda_{i,j},m}$ to the remote label $sl_{\lambda_{i,j},m'}$.

Let event e_r cause that job $\phi_{\lambda_{i,j}}$ reads its inputs. The reading instant of job $\phi_{\lambda_{i,j}}$ is measured in the local time $t_{Z_a}(e_r)$ of memory $m \in Z_a$, such that

$$t_r = t_{Z_a}(e_r).$$

The occurrence of the writing event e_w is also measured in local time $t_{Z_b}(e_w)$ of the memory $m' \in Z_b$

$$t_w = t_{Z_b}(e_w).$$

The constant time interval between the occurrence of the reading event e_r and the writing event e_w is called logical execution time $LET_{\phi_{\lambda_i}}$.

We would like to shortly discuss the impact of the synchronization error ϵ on communication times between different time zones. Assume $\phi_{\lambda_{i,j}}$ copies a value from label $sl_{\tau_{i,j},m}$ in Z_a to the distant label $sl_{\tau_{i,j},m'}$ in Z_b. Due to the synchronization error, the value cannot safely be expected before $t_{Z_a}(e_r)$ + max. comm. delay + ϵ in time zone Z_b. Thus a safe lower bound for the LET of a LET interconnect task ϕ_{λ_i} is given by $LET_{\phi_{\lambda_i}} \leq$ max. comm. delay + ϵ. Note that Definition 16 allows that a LET interconnect task may cross several time zones without local clock synchronization. This "fly over" avoids additional synchronization latencies.

B. Properties

In this section, we show that the important properties of the classical LET programming model are still valid in the more general context of the SL LET programming model.

1) Time determinism: Time determinism as specified in Def. 8 is a task property stating that the timing of task outputs is identical in any execution run. In contrast to BET tasks, LET tasks have this property [12] since they are activated at deterministic points in time and terminate after a static logical execution time independent of the momentary workload. The transition from the component level to the system level LET programming model implies that a LET task resides in a time zone. LET tasks are thus time-deterministic in local time. LET interconnect tasks, communicating a value from time zone

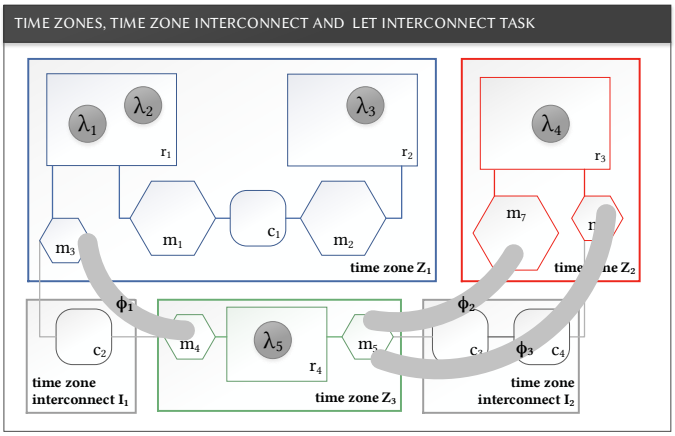


Fig. 4: Illustrating new concepts for applying LET at the system level

Z_a to time zone Z_b, produce a time-deterministic output in local time of Z_b. This is due to their deterministically timed activation in a time zone Z_a and their completion in time zone Z_b with the elapse of the static logical execution time (which should account for the maximum synchronization error as explained above). Since local times are approximates of a global time, LET tasks and LET interconnect tasks are also time-deterministic in global time with a bounded error.

2) Data flow determinism: Data flow determinism as a property (Def. 9) refers to a pair of communicating tasks $\tau_k \prec \tau_i$, where τ_k provides data to τ_i. If data flow determinism applies, each consumer job $\tau_{i,j}$ reads a value always from the same producer job $\tau_{k,l}$ in any execution run. We reason briefly why this is true in the context of the LET programming model. Two LET jobs can communicate if the lifetime of the label written by the producer job overlaps with the read phase of the consumer job [14]. LET tasks are time deterministic and label lifetimes of producer jobs of the same task are distinct (without overlap) if some overwriting semantics is applied. Furthermore, the read interval of a LET job condenses to a point in time. Thus there is exactly one possible intersection of the read phase of a consumer job with the label lifetime of a producer job, which implies that any pair of communicating LET tasks is data flow deterministic.

We proceed now to the SL LET programming model. Assume that a LET task λ_i is the producer task in a time zone Z_a and a LET interconnect task ϕ_{λ_i} is the consumer task. Then still all requirements for data flow determinism hold in the relevant time zone Z_a, since λ_i provides a time deterministic output in Z_a and ϕ_{λ_i} reads inputs periodically in zero time in Z_a. Conversely, assume now that the LET interconnect task ϕ_{λ_i} is the producer task and an LET task λ_j is the consumer task in a time zone Z_b. Here ϕ_{λ_i} is time deterministic in Z_b producing an output which is read periodically by λ_j in zero time. For both cases, there is only one possible intersection of a label lifetime with the read phase of a consumer job in the local time.

3) Composability: LET tasks are composable w.r.t. response times given the classical LET programming model [12]. With the introduction of SL LET, LET tasks can be added to different resources in different time zones and LET interconnect tasks can be added to communicate values between time zones. We consider now a LET task in a given time zone Z. Inserting another LET task in the same time zone Z does not affect composability, since the rules of classical LET programming model still apply in this scope. The addition of LET tasks in remote time zones is transparent to the LET task in Z. Only with the introduction of a LET interconnect task, which links time zones Z and Z', an additional workload appears in time zone Z. A schedulability test is required to verify that the LETs of all LET tasks in Z are still time-safe. If so, then the timing of LETs tasks in Z are unaffected by the addition of the LET interconnect task and response times are composable. Beyond that, SL LET is not only composable w.r.t. response times but also w.r.t. label lifetimes, which are individual design decisions for each time zone.

V. System Level LET in the Design Process

The programming model extension described in this paper fits the automotive design process. It exploits the typical separation of network and ECU design.

Local multicore ECU design with the LET programming model can remain as today with a chip plus memory as a single time zone. All computation only uses local LET labels. The read and write stubs for LET interconnect tasks across chip boundaries can already be implemented. ECUs consisting of multiple processor chips can form a single time zone or, if latencies grow larger, can be split in several time zones. Different ECUs are likely to be in independent time zones which is acceptable because they will be designed by different design teams requiring defined interface specification. In any case, due to the communication task properties, time zones can be defined and modified at any time in the design process as long as all communication between time zones uses LET interconnect tasks.

ECU and the related function specification provide an estimation of the main traffic streams to network design often sufficient to identify network bottlenecks and use timing analysis for an estimation of communication latencies. Once available, the communication task requirements can be used for network latency specification and validation. The communication tasks not only support message passing communication where communication is explicit but also shared memory communication effectively providing a platform communication service. Communication service implementation can be separated from application which increases portability. In essence, task assignment is coupled to a variable-to-zone binding process. Such binding can easily be modified. With this property, the System Level LET extensions support important design use cases.

- *Adding a task* is a use case for composability. All LET labels of the respective time zone are available to the new task. Changes to the response time are local only.
- *Task remapping* to an ECU in a different time zone, tasks will access the LET label in the new time zone. Both affected ECUs must be re-analyzed locally. If the label is not available in then new time zone, communication must be adapted.
- *Porting an application* to a new platform requires the availability of a communication service that provides the required communication tasks on the new platform. This is a clearly defined platform design task that can be developed independent of the application.

VI. Use Case

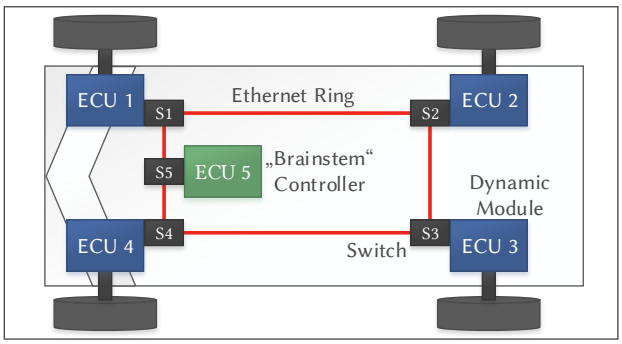
In this section, we provide an automotive use case example and describe how it can benefit from the concept of SL LET. The increasing demand for communication bandwidth, flexibility and optional security enforcement supports the trend towards Ethernet based in-vehicle networks. In contrast to the classical bus based topologies like CAN or FlexRay, Ethernet provides the ability to construct heterogeneous topologies with point to point connections. As a drawback, end-to-end communication latencies may vary considerably for each sender-receiver path.

The use case is inspired by an experimental vehicle of the project "MOBILE" [15]. This fully electrical car is an example for future vehicles with independent control of steering, drive and brake per wheel and many concepts are adapted for upcoming autonomous driving projects like UNICARagil [16]. Figure 5a shows a communication topology with four independent dynamic modules (ECU 1-4) and a "brainstem" controller (ECU 5). All ECUs are connected to an Ethernet backbone, that has to handle not only latency-critical control traffic but also bandwidth demanding sensor traffic (e.g. additional camera or LIDAR). The Ethernet ring topology has important benefits like low wiring costs and less complex switches, reducing the overall integration costs compared to a star or tree topology. Figure 5b shows the functional model of an Electronic Stability Program (ESP), which takes as input a set trajectory, the wheel speeds and a yaw rate. As outputs, the ESP provides individual brake, drive and steer values for each wheel. Sensing the wheel states and controlling the wheel brakes (red path in Figure 5b), creates bidirectional traffic between ECU 5 and ECU 1-4. Without mechanical coupling between the wheels, communication jitter becomes critical for correct control. Even slightly asynchronous controlled wheels (e.g. steer angle and brake) destroy the effect of an ESP. Therefore, it is crucial to reduce communication jitter as much as possible.

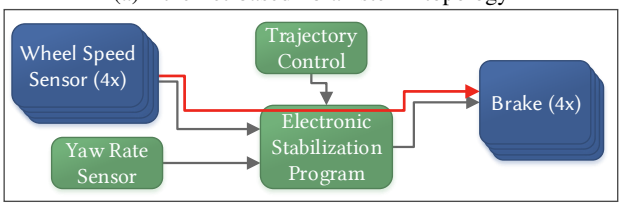
A. Evaluation of a cause-effect chain

In the following, we investigate an exemplary cause-effect chain of the ESP. We compare three different approaches w.r.t. end-to-end timing properties and implementation overhead: (1) The conventional LET applied to non-zero communication times, (2) conventional LET at ECU level and BET communication in-between, (3) SL LET.

The example in Figure 6 illustrates a distributed cause-effect chain from a wheel (ECU 2) to the ESP controller (ECU 5)



(a) Ethernet based “brainstem” topology



(b) Functional description of a distributed ESP system

Fig. 5: Automotive use case

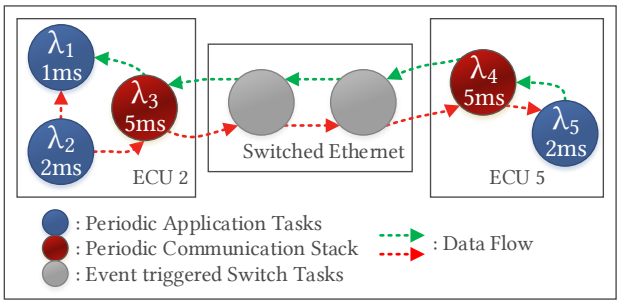


Fig. 6: Exemplary ESP cause-effect chains

and back. It consists of three periodic application tasks and two communication stack tasks. As it is common in automotive designs, each communication stack is implemented as a periodic task, here 5ms. To model the behavior of switched Ethernet, event-triggered tasks are used. Both ECUs employ a rate monotonic scheduling and we assume that they are sufficiently synchronized (e.g. $\epsilon = 100\mu s$). A timing diagram is shown in Figure 7. Techniques for time synchronization over Ethernet with precision time protocol (PTP) are standardized in IEEE 802.1As [17].

1) Conventional LET with non-zero time communication: CASE 1 in Figure 7 shows why it is not sufficient to extend the LET of a task such that it includes now also non-zero communication delays. We focus on the cause-effect chain segments $\lambda_2 \prec \lambda_1$ and $\lambda_2 \prec \lambda_5$. While the output of λ_2 is instantly available inside ECU 2, the communication delay to ECU 5 is considerable. As a result, the LET of λ_2 has to be far larger than its period. This means that the initial and important assumption $LET \leq period$ of the conventional LET paradigm is violated.

2) Conventional LET within ECUs and BET communication in-between ECUs: As a consequence of the previous modeling attempt, one may consider to use LET within an ECU scope and BET communication in-between (CASE 2). Figure 7 extends the previous example by adding the return data flow from λ_5 to λ_1, i.e., the sending process of the actuator values to the wheel. To simplify the timing diagram, the communication stacks use an early-write strategy. This means that they write shortly after their worst case response times (WCRTs), resulting in smaller end-to-end delays. Figure 7 shows that the problem which was originally solved by LET, namely eliminating the unpredictable job-level precedences, now reappears on a global scale due to varying communication delays. E.g., the jobs $\lambda_{2,1}$ to $\lambda_{2,4}$ are all potential predecessors of $\lambda_{5,8}$. The return segment of the cause-effect chain aggravates the non-determinism in data flow, creating a multitude of possible end-to-end data flow paths. Transferring a timestamp with every message is not an option, since typical control algorithms rely on sequential data and do not compensate data age variations during runtime.

3) SL LET: With SL LET as CASE 3, this non-determinism in data flow can be removed. We start by applying the above presented SL LET concepts to the example in Figure 6. Time zones correspond to scopes in which zero-time communication is possible. In this example, each ECU has its own time zone and the time zone interconnect is the switched Ethernet. There are two LET interconnect tasks which realize the communication between $\lambda_2 \prec \lambda_5$ and $\lambda_5 \prec \lambda_1$ respectively. They comprise the involved communication stack tasks and the switch tasks. The LET of each interconnect task corresponds to the maximum latency derived under the BET model for the respective communication path. Therefore, CASE 2 and CASE 3 have identical worst case latencies. Depending on the requirements, different versions of label management are possible. In the common case, a simple overwrite policy is applied. SL LET improves here over inter-ECU BET communication, since data flow paths are now deterministic. E.g., job $\lambda_{5,8}$ will always read from job $\lambda_{2,1}$. Consequently, the jitter is eliminated – which was an important design goal. However, data aga inconsistencies compared to a single ECU implementation may still occur. The basic problem was illustrated in Figure 3. This can be addressed by an appropriate label management.

The implementation overhead for SL LET is comparable to the conventional LET concept, that has already been accepted by industrial practice [3]. Additional buffers are required if over-writing – which we consider as the standard case – is not used for label management. Single assignment labels are only of interest if a high consistency in data age is required for a specific cause-effect chain. The number of necessary buffers in this case depends on the tolerated difference in data age among a set of inputs. Since only the receiver task is responsible for label management while the sender task is agnostic of it, the approach scales and is compositional. Note that with SL LET, over sampling can be eliminated in design since precedences are enforced and known. This counteracts

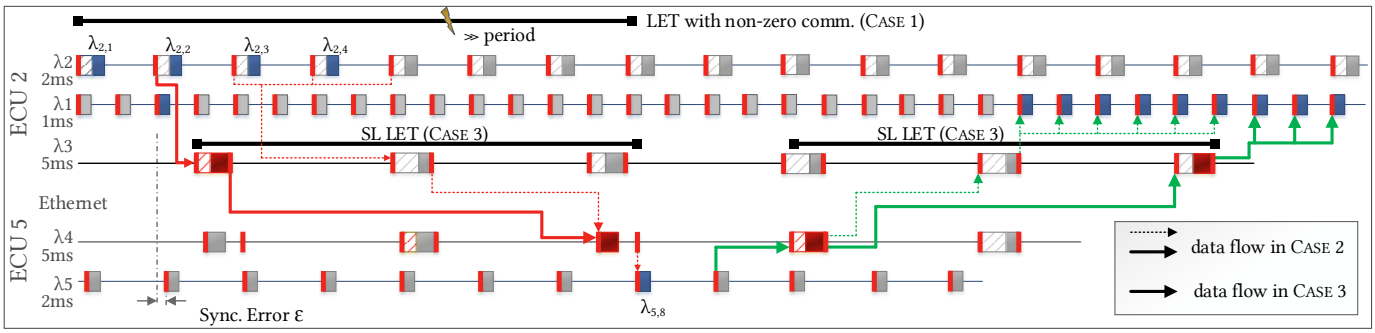


Fig. 7: Timing diagram for (1) LET with non-zero time communication, (2) LET at ECU level and BET communication between ECUs, (3) SL LET *The examples assumes that the delay regarding Ethernet transmission varies between* 0 *and* $5ms$.

the habit in automotive industry to send data with a higher frequency than needed, to maintain control quality when data age varies.

VII. Related Work

The Logical Execution Time (LET) abstraction [18] was introduced as a real-time programming paradigm. LET abstracts from the actual timing behavior of real-time tasks on the physical platform and thus introduces a separation between functionality on the one hand, and mapping and scheduling on the other hand. It also provides a clean interface between the timing model used by the control engineer and that of the software engineer. LET was introduced with the time-triggered programming language Giotto [19], followed by HTL (for Hierarchical Timing Language) [20] and TDL (for Timing Definition Language) [21]. LET-oriented runtime systems include the E machine [22] and Variable-Bandwidth Servers (VBS) [23].

The LET paradigm was considered until recently by the automotive industry as not efficient enough in terms of buffer space and timing performance. The shift to embedded multicore processors has represented a game changer: The design and verification of multicore systems is a challenging area of research that is still very much in progress. Predictability clearly is a crucial issue which cannot be tackled without changes to the design process [24]. Several OEMs and suppliers [25] [26] have come to the conclusion that LET might be a key enabler and a standardization effort is already under way in the automotive community to integrate LET into AUTOSAR.

Loosely time-triggered architectures (LTTA) [27] are related to SL LET, but they are only defined for time-triggered local scheduling. LTTA uses "Communication by Sampling (CbS)" where data are communicated with independent data sampling, communication and writing between non-synchronized domains with independent clocks ($\epsilon \to \infty$). As it permits clock drift (deviating clock frequencies), data losses generally cannot be avoided. This is a strong limitation when extending LET to the system level. Since all tasks including communication are time-triggered, "fly over" is not possible. LTTA uses a register semantics avoiding buffer overflow while back pressure LTTA [28] implements an "elastic" circuit based on event driven Kahn Graphs, and is therefore not compatible with the LET model. In contrast, SL LET can use any communication mechanism including combinations of time triggered and event driven communication as used in most automotive networks, as long as the label values are eventually delivered. This is possible because SL LET makes the essential assumption of bounded clock drift and jitter.

The LET abstraction is related to the synchrony hypothesis of synchronous reactive programming [29] and to real-time scheduling theory [30]. The synchrony hypothesis makes the simplifying assumption that the program reacts instantaneously to its changing environment. Synchronous programs are known to be difficult to parallelize due to the need to resolve through fixed-point analysis signal statuses and causality issues. Thus, concurrency is typically compiled away to produce sequential code. In LET programming reading input and writing output is cycle-free so no fixed-point analysis is needed. Note that the LET paradigm and the synchronous abstraction have in common the objective to provide a platform-independent programming abstraction for real-time systems.

VIII. Conclusion

The LET programming model was successfully used to master automotive multicore designs. This has spurred interest in a wider use of LET, but the inherent synchronicity requirements prevent application to distributed systems. The paper introduced System Level LET (SL LET) as an LET extension with time zones and communication tasks as main features. SL LET can be used in mixed programming model environments with a variety of scheduling algorithms thereby keeping the essential LET properties of composability, time determinism and data flow determinism under clearly defined conditions. SL LET also fits the current approach of separating network design from ECU design. While the paper focused on automotive applications, SL LET can be used in any design with minimum clock synchronization as defined in the paper. 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.

REFERENCES

[1] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[2] D. Ziegenbein and A. Hamann, "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*, 2015, pp. 56:1–56:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2747947

[3] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger, "Towards parallelizing legacy embedded control software using the let programming paradigm," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*. IEEE, 2016, pp. 1–1.

[4] S. Resmerita, A. Naderlinger, M. Huber, K. Butts, and W. Pree, "Applying real-time programming to legacy embedded control software," in *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*. IEEE, 2015, pp. 1–8.

[5] S. Kuntz, "Multicore and Logical Execution Time, Industry's Perspective," *Dagstuhl Seminar 18092 – The Logical Execution Time Paradigm: New Perspectives for Multicore Systems*, 2018.

[6] R. Mader, "Implementation of Logical Execution Time – in an AUTOSAR based embedded automotive multi-core application," *Dagstuhl Seminar 18092 – The Logical Execution Time Paradigm: New Perspectives for Multicore Systems*, 2018.

[7] A. consortium, "Autosar timing extension," https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TPS_TimingExtensions.pdf, 2018.

[8] M. Beckert and R. Ernst, "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)*, March 2018.

[9] A. Biondi, P. Pazzaglia, A. Balsini, and M. Di Natale, "Logical execution time implementation and memory optimization issues in autosar applications for multicores."

[10] R. Ernst, S. Kuntz, S. Quinton, and M. Simons, "Dagstuhl Seminar 18092 – The Logical Execution Time Paradigm: New Perspectives for Multicore Systems," https://www.dagstuhl.de/no_cache/en/program/calendar/semhp/?semnr=18092, 2018.

[11] A. consortium, "Autosar standards," https://www.autosar.org/standards/, 2018.

[12] C. M. Kirsch, "Principles of real-time programming," in *International Workshop on Embedded Software*. Springer, 2002, pp. 61–75.

[13] M. Panunzio and T. Vardanega, "On component-based development and high-integrity real-time systems," in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 79–84.

[14] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Analyzing end-to-end delays in automotive systems at various levels of timing information," in *IEEE 4th International Workshop on Real-Time Computing and Distributed systems in Emerging Applications (REACTION)*, 2016.

[15] P. J. Bergmiller, *Towards functional safety in drive-by-wire vehicles*. Springer, 2015.

[16] "Unicaragil project website," http://www.unicaragil.de/, accessed: 2018-03-26.

[17] "Ieee 802.1as - timing and synchronization," http://www.ieee802.org/1/pages/802.1as.html, accessed: 2018-05-30.

[18] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*, 2012, pp. 103–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24349-3_5

[19] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[20] T. A. Henzinger, C. M. Kirsch, E. R. B. Marques, and A. Sokolova, "Distributed, modular HTL," in *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, 2009, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2009.9

[21] W. Pree and J. Templ, "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*, 2006, pp. 133–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70930-5_9

[22] T. A. Henzinger and C. M. Kirsch, "The embedded machine: Predictable, portable real-time code," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, p. 33, 2007. [Online]. Available: http://doi.acm.org/10.1145/1286821.1286824

[23] S. S. Craciunas, C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "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*, 2009, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1109/SIES.2009.5196213

[24] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi, "Building timing predictable embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4, pp. 82:1–82:37, 2014. [Online]. Available: http://doi.acm.org/10.1145/2560033

[25] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle, "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*, 2014, pp. 53–62. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2014.28

[26] F. Walkembach, "Model-driven development for safety-critical software components. White paper," 2016. [Online]. Available: http://events.windriver.com/wrcd01/wrcm/2016/08/WP-model-driven-development-for-safety-critical-software-components.pdf

[27] A. Benveniste, "Loosely time-triggered architectures for cyber-physical systems," in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, 2010, pp. 3–8. [Online]. Available: https://doi.org/10.1109/DATE.2010.5457246

[28] S. Tripakis, C. Pinello, A. Benveniste, A. L. Sangiovanni-Vincentelli, P. Caspi, and M. D. Natale, "Implementing synchronous models on loosely time triggered architectures," *IEEE Trans. Computers*, vol. 57, no. 10, pp. 1300–1314, 2008. [Online]. Available: https://doi.org/10.1109/TC.2008.81

[29] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[30] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, ser. Real-Time Systems Series. Springer, 2011, vol. 24. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-0676-1