Industry-track: System-Level Logical Execution Time for Automotive Software Development

Kai-Björn Gemlau
TU Braunschweig
Braunschweig, Germany
gemlau@ida.ing.tu-bs.de

Hermann v. Hasseln
Mercedes-Benz AG
Sindelfingen, Germany
hermann.v.hasseln@mercedes-benz.com

Rolf Ernst
TU Braunschweig
Braunschweig, Germany
ernst@ida.ing.tu-bs.de

***Abstract*—The way how automotive software is developed has rapidly evolved with the introduction of heterogeneous hardware/software architectures. Nevertheless, the requirement for deterministic behavior of safety-critical cause-effect chains persists unchanged. As a side effect of the shared platform, complex dependencies between critical and non-critical functions arise, demanding a model-based approach to handle time determinism throughout the design process. Limited to the scope of a single component, the Logical Execution Time (LET) paradigm provides such an abstraction of the runtime behavior. It has been successfully introduced in AUTOSAR to mitigate the design complexity, ensure a deterministic timing behavior and facilitate a lock-free communication. This paper discusses how the scope of LET can be extended to the system level, enabling an efficient design of distributed AUTOSAR software, where robustness towards platform changes plays a key role. System-Level Logical Execution Time (SL-LET) is currently in the process of AUTOSAR standardization, supported by a joint group of industry and academic partners.**

***Index Terms*—AUTOSAR, System-Level LET, data-flow determinism**

I. Challenges in Automotive Software Development

Modern cars have continuously evolved into data-centric software systems that are implemented on top of heterogeneous and distributed hardware architectures. While the number of electronic control units (ECUs) in a car has increased over the last decades, such a fully distributed architecture has reached its limits in terms of maintainability, energy efficiency and interconnect complexity. Consequently, there is the trend towards centralized and zonal architectures, where a few high performance ECUs provide a shared platform for all different kinds of automotive software, each bringing its own requirements in terms of performance, data exchange and safety. Moreover, automotive software can not be seen as a static system, but frequent release cycles and partial software updates become the rule rather than the exception [1]. This poses a great integration challenge on original equipment manufacturers (OEMs).

To handle the design complexity in a model driven development (MDD) process, AUTOSAR has developed two types of platforms in the past, namely the AUTOSAR Classic and the AUTOSAR Adaptive platform. The Classic platform addresses the needs of traditional control functions with high safety demands, such for example powertrain and break control. It typically builds on top of a real-time operating system (RTOS) with static configuration. The Adaptive platform on the other hand provides a service-oriented architecture for great flexibility on top of a POSIX compliant operating system.

Although the platform has evolved, fundamental safety requirements remain the same. The developer of a complex vehicle function has to deal with three major challenges. First, the functional model shall be platform agnostic and a high degree of platform independence shall be retained throughout the development process. This is important, since function shall be applicable to a large number of product variants, which itself poses great challenges in the development [2]. Second, safety requirements are posed on the whole function and need to be decomposed in timing requirements for the resulting cause-effect chain. This chain ranges from the periodic sampling and fusion of sensor data for environment perception over the planning stage to the control and actuation of a car. The data-flow as well as the timing within this cause-effect chain has to be kept consistent between the specification and the implementation. Last but not least, the function chain shares the distributed platform with many other functions, each introducing its own requirements and dynamic behavior. Such a mixed-criticality system as well as the involved high-performance architectures lead to timing interference and timing uncertainty.

In the remainder of this paper, we discuss what kind of specification method can aid the function developer and the integrator to design complex distributed systems, without foreseeing or unintentionally restricting the design space of the implementation. With the LET paradigm [3], a promising concept has already been established and widely adopted for the development of non-distributed functions. This is described in Section II. Section III discusses what new requirements arise when the LET approach shall be extended to the system level and finally Section IV describes the resulting SL-LET paradigm and its current status in AUTOSAR standardization.

II. Gaining Determinism: Logical Execution Time

The LET paradigm conceptually separates computation from communication. Without LET, a periodic task reads its input at the beginning and writes the output at the end of

Thanks to the AUTOSAR SL-LET working group for the valuable discussions.

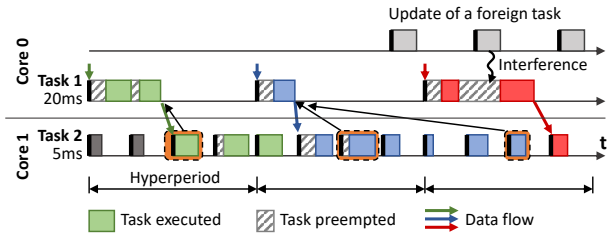

Fig. 1. Missing data-flow determinism without LET

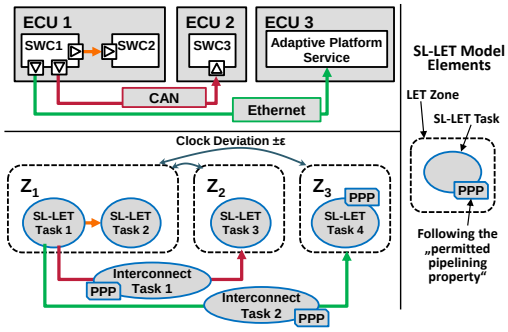

Fig. 2. SL-LET model of a distributed system

its execution. This has been the prevalent model for real-time tasks. As a result, jitter in the execution time may affect the data-flow between tasks such as given in Figure 1. The colors green, blue and red denote the data-flow between the 20ms task and the 5ms task. While the third job of the 5ms task in the first two hyperperiods reads fresh data (from the same hyperperiod), it reads older data (the blue sample) in the last hyperperiod. This non-deterministic data flow is caused by an update of a foreign task, which induces interference on the 20ms task. While this is already problematic in a single implementation, it becomes urgent in case of dynamic architectures. An update or modification in a completely different function may, due to the shared platform, cause timing interference and affect the execution time jitter of the task.

LET provides a logical wrapper for the execution of a real-time task [3]. Instead of communicating at the borders of the physical execution, the task now communicates at two distinct points in time. Those events delimit the logical LET interval. The physical execution must then take place anywhere within the LET interval. This abstraction has multiple benefits. First, it provides a deterministic data flow, as long as the implementation can guarantee that the execution fits in the LET interval [3], [4]. At the same time, it is a very simple specification mechanism that is agnostic of any platform dependent effects such as execution time jitter. It provides the function developer with a handy tool to specify the data flow in a cause-effect chain, while providing a large degree of freedom for implementation. Thereby LET allows an explicit notion of robustness against a modified runtime behavior [5], which aids a design with a higher degree of platform independence. Depending on the application requirements, dimensioning of the LET interval can be done based on analytical or experimental worst-case estimations, but in both cases a run-time monitoring can be established with low overhead [6]. Especially important for modern data-centric systems is the ability to design lock-free communication, which has successfully been exploited in multicore ECU design [7], e.g., integrated in the electric vehicle family EQ of Mercedes-Benz AG. Although LET is a form of timed programming, it must not be mistaken with time driven scheduling like time-division multiple access (TDMA). LET does not enforce a specific scheduling strategy, wherefore work-conserving scheduling can be applied [8], [9]. Due to those benefits, LET has been introduced in the AUTOSAR timing extensions [10].

III. LET on System Level: New Requirements

The ability to use LET throughout the whole development process is promising, although LET comes with some fundamental limitations. First, the length of the LET interval is conceptually bounded by its period. An LET interval larger than its period would imply, that there may be multiple jobs of this task executing in parallel. Without further constraints, this is like re-executing a finite state machine before it has created its output, which may produce non-deterministic results. To be applicable to a functional model or distributed communication, where the latency likely exceeds the period, exactly such large LET intervals have to be taken into account. Second, LET does not comprise the notion of distributed clocks. For any distributed platform, perfect clock synchronization is not implementable, but the different clocks may have a bounded synchronization accuracy.

Due to those limitations, LET has been restricted to the ECU's scope. Besides overcoming those two constraints, a system-level extension of LET shall **(A)** be applicable to all stages of the model-driven development process, **(B)** provide a logical abstraction of significant communication latencies (comparable to the abstraction of computation in LET) and **(C)** evolve existing LET benefits, particularly the lock-free communication, to the system level.

IV. System-Level Logical Execution Time

SL-LET [11] is the consistent further development of the LET idea for the system level. The concept comprises three main elements, which are shown in Figure 2: **(A)** The Permitted Pipelining Property (PPP) allows to capture systems, where the SL-LET interval is larger than its period. It explicitly states that a pipelined execution of the related task is permitted, which holds if no data from an unfinished preceding execution is accessed. This includes the important class of concurrent execution of stateless tasks, e.g., on parallel hardware. This optional property can be assigned already to coarse-grained blocks in the functional model and inherited in further development phases. If pipelined execution is not explicitly permitted, SL-LET intervals larger than the period have to be decomposed during the development steps until the resulting intervals are smaller than their period. This ensures that there is no inconsistency during later integration. **(B)** LET zones

support the meaning of SL-LET events in a distributed system, by providing a common understanding of time respectively clocks and their synchronization accuracy. A LET zone models a subsystem with a local time base that has a bounded clock deviation to other LET zones. Any time instant within one LET zone can therefore be translated in a time interval (twice the size of the synchronization accuracy) in another LET zone. **(C)** Based on this common understanding of time, SL-LET intervals are delimited by two SL-LET events, each associated with a LET zone and therefore a specific time base. Interconnect tasks generalize over LET tasks by allowing that the two labels may be in different LET zones. This allows to extend the LET idea for a distributed system. For larger time deviations, interconnect tasks must have PPP, which naturally holds for many message-passing interconnects like busses/networks (inter-ECU) as well as message-queues (intra-ECU).

Working with SL-LET closely follows the approach of repeating composition and decomposition, which is already known in the remaining development process. Like a vehicle function is decomposed to a function chain and later on translated to software components, SL-LET follows the same decomposition for the timing model. It thereby allows to decompose the timing requirements while preserving the data flow. On the other hand, existing non-distributed software, which was originally described with LET, can be encapsulated in a SL-LET wrapper and re-used seamlessly.

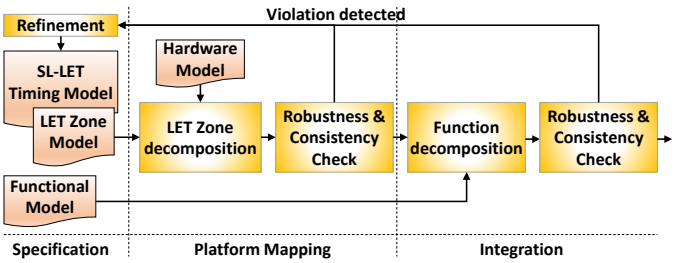

Fig. 3. Development process with SL-LET

Figure 3 provides an overview on how SL-LET can be utilized in the development process. The key strength of SL-LET is the robustness, which can automatically be checked throughout the process. The initial functional model remains platform agnostic and the corresponding SL-LET specification may assume a perfect clock synchronization. Nevertheless, it already contains an inherent robustness towards a worst-case clock synchronization accuracy. This becomes apparent by analyzing which synchronization accuracy would violate the intended data flow, analogue to [4]. Combined with a specific hardware model, LET zones can be decomposed and the adherence to the robustness can be checked. This decomposition and model checking can be continued during integration. A model refinement and therefore a potential costly design iteration is not needed until a robustness (either latency or synchronization accuracy) is exhausted. During the whole process, a clear tracing between implementation properties and the specified safety/timing requirements is possible, supporting the MDD idea [6].

The benefits of SL-LET in the development of automotive software clearly justify its introduction in AUTOSAR. The simplicity of SL-LET allows to cover both, the AUTOSAR Classic as well as the AUTOSAR Adaptive platform and the standardization is currently in progress, supported by a joint working from both, industry and academic partners.

V. Conclusion

In this paper, we have shown the need for an easy-to-use time programming paradigm, which can be efficiently applied throughout the different stages of automotive software development. We think that SL-LET is able to provide such a unified view on timing, as it allows to incorporate communication as well as the notion of deviating time bases in a distributed system. Thereby it enables an efficient and lock-free implementation of communication. SL-LET follows the approach of repeating composition and decomposition, which make it easy to integrate in the remaining software development process. Although the specification itself is simple, it allows for an efficient monitoring of robustness against platform changes, both, at design time and at runtime. This underlines its relevance for future automotive systems, where updates become the rule rather than the exception.

References

[1] D. Claraz, R. Mader, H. von Hasseln, and M.-J. Friese, "A dynamic Reference Architecture to achieve planned Determinism for Automotive Applications," in *ERTS 2022*, Toulouse, France, Jun. 2022.

[2] D. Strüber, M. Mukelabai, J. Krüger, S. Fischer, L. Linsbauer, J. Martinez, and T. Berger, "Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, ser. SPLC '19. New York, NY, USA: Association for Computing Machinery, Sep. 2019, pp. 177–188.

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

[4] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, pp. 104–113, 2017.

[5] Institute of Computer and Network Engineering, TU Braunschweig, "TORO (Analysis TOol to evaluate the latencies and RObustness of cause-effect chains)," https://github.com/IDA-TUBS/TORO, 2022.

[6] M. Möstl, "On Timing in Technical Safety Requirements for Mixed-Critical Designs," Ph.D. dissertation, TU Braunschweig, 2021.

[7] 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 *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Soc., 2016, pp. 1–1.

[8] A. Biondi, P. Pazzaglia, A. Balsini, and M. Di Natale, "Logical execution time implementation and memory optimization issues in autosar applications for multicores," in *Proc. of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2017)*, 2017.

[9] 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)*, 2018.

[10] *AUTOSAR - Specification of Timing Extensions*, R21-11 ed., AUTOSAR GbR, Nov. 2021.

[11] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-time Automotive Software," *ACM Transactions on Cyber-Physical Systems*, vol. 5, no. 2, pp. 14:1–14:27, Jan. 2021.