2025 Design, Automation & Test in Europe Conference (DATE 2025)
Special Initiative "Autonomous Systems Design"

Modeling the SL-LET paradigm in AUTOSAR Adaptive

Davide Bellassai[*◇], Gerlando Sciangula[*◇], Claudio Scordino[◇], Daniel Casini[*], Alessandro Biondi[*]
[*]Scuola Superiore Sant'Anna, Pisa, Italy
[◇]Evidence S.r.l., Pisa, Italy

***Abstract*—The AUTOSAR consortium proposed the AUTOSAR Adaptive standard to tackle the challenges introduced by the design of modern automotive systems. It consists of a service-oriented architecture (SoA) implemented in C++ and built on top of POSIX operating systems. However, unlike the previous AUTOSAR Classic specifications, this novel standard does not address non-functional requirements, including determinism, which is of key importance to guarantee the system's functional safety. This paper proposes a modeling extension to the AUTOSAR Adaptive standard aiming at guaranteeing a deterministic execution by leveraging the System-Level Logical Execution Time (SL-LET) paradigm, already used in the context of AUTOSAR Classic. A prototype implementation is also proposed, which is used to experimentally corroborate the feasibility of the proposed model extension with an evaluation based on a realistic automotive application built on the official AUTOSAR Adaptive Platform Demonstrator (APD).**

***Index Terms*—real-time, automotive, AUTOSAR standard, Logical Execution Time, modeling.**

I. Introduction

The automotive industry is facing a revolution to support more complex functionalities needed by autonomous and assisted driving systems. Consequently, automotive functionalities are moving from systems based on electronic functionalities made of simple Electronic Control Units (ECUs) executing ad-hoc operating systems (e.g., OSEK [1]) and communicating through domain-specific networks (e.g., CAN, LIN, FlexRay) to POSIX-based operating systems and Ethernet networks. This paradigm shift also implied a change in the standards: while classical automotive systems are based on the AUTOSAR Classic standard [2], the need for new and more complex functionalities has forced the automotive industry to create the novel AUTOSAR Adaptive standard [3], which overtakes the original design.

This new set of specifications has borrowed concepts and paradigms from the widely used POSIX standard, creating a service-oriented architecture (SoA) to be run on modern multi-core heterogeneous platforms. However, unlike the previous AUTOSAR Classic standard, AUTOSAR Adaptive is lacking some key support to guarantee non-functional properties (e.g., determinism), preventing the system from guaranteeing the timing constraints of the executed tasks.

This paper illustrates how the well-known System-Level Logical Execution Time (SL-LET) paradigm [4], already used in AUTOSAR Classic [5] to achieve deterministic communication among tasks, can be modeled in the AUTOSAR Adaptive platform as well. In particular, the paper aims to lay the foundations for the standardization of SL-LET in the AUTOSAR AP specifications. We believe that the introduction of deterministic communication can help in supporting functional safety in this new class of systems.

II. Background

A. Architecture of AUTOSAR Adaptive

AUTOSAR (AUTomotive Open System ARchitecture) is an international consortium born in 2004 to create a standard and interoperable software architecture for automotive ECUs. The AUTOSAR Classic [2] specification provides a reference model and programming API for ECUs executing in a real-time operating system and signal-oriented communications.

Later, the consortium proposed the AUTOSAR Adaptive standard, which specifies a modern service-oriented architecture (SoA) built on top of POSIX operating systems [6]. The Adaptive platform is based on different *Functional Clusters*, which provide services and functionalities available to developers through a C++ API, called *Runtime Environment for Adaptive Applications (ARA)*. Figure 1 shows the architecture of the Adaptive Platform and the services offered by ARA. This work primarily focuses on the *Communication Management* cluster, responsible for service-oriented communication, which leverages the SOME/IP [7] communication protocol, which is highly recommended by the standard. However, the work can be easily adapted to the DDS protocol, which is receiving a growing interest from the automotive industry [8].

The foundation of message-passing communication between software components (SWCs) is the client-server pattern, in which servers provide *services* that clients may request, such as: **(i)** *Events*: let clients know about server-side events; **(ii)** *Methods*: let servers provide remote procedure calls (RPC) available for clients to use; and **(iii)** *Fields*: give clients access to retrieve or set remote data. Specific ARXML files are used to describe services' interfaces at design phase, while service discovery is used at run-time to bind clients and servers.

B. System Level Logical Execution Time

The Logical Execution Time (LET) paradigm was introduced as part of the GIOTTO framework [9], and it is commonly used to achieve deterministic communication of

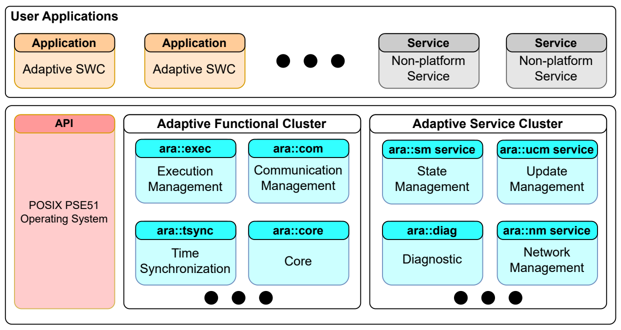

Fig. 1. Architecture of AUTOSAR Adaptive [3].

automotive applications made of periodic tasks [10]. In classical LET approaches for intra-ECU multi-core platforms, the jitter of communication, which depends on the scheduling of tasks, is canceled by forcing input/output operations to occur at pre-defined time instants — most commonly, at tasks' periods boundaries. Furthermore, at each of such instants, all outputs must occur before any input.

The original LET paradigm assumed a negligible time for input and output operations [11], [12]. Although this assumption could be acceptable in the specific case of shared memory communications, it becomes unrealistic for message-passing communications, which could require the execution of context switches (e.g., to enter kernel space), complex stacks (e.g., TCP/IP), and incur in network delays. For this reason, more recently, SL-LET [4] has been introduced as an extension to achieve deterministic execution in distributed systems as well. The SL-LET introduces the concepts of: **(i)** *timezones*, which represent the scope of a local time for each ECU, and **(ii)** *interconnect task*, whose purpose is to manage communications between different ECUs, hiding the unpredictable network delay under its logical execution time. The difference between two different time zones is bounded by a synchronization error.

III. Model and Design

We present a possible approach to integrate the SL-LET paradigm into AUTOSAR Adaptive. Our design principle aimed at maintaining the original philosophy and architecture of the standard whilst minimizing the changes to the API exposed to the system programmer. We first present the SL-LET protocol that fits with the Adaptive Platform design in Section III-A, introducing different possible approaches able to guarantee deterministic execution. Then, we present an extension to the standard model to integrate the SL-LET paradigm into AUTOSAR Adaptive Platform (Section III-B). Finally, we report an example of ARXML configuration file (Section III-C), including all the required items to enable the SL-LET usage.

A. SL-LET protocol

The protocol adopted in this work is designed to address the specific needs of the AUTOSAR Adaptive communication environment, focusing on providing deterministic communication for communicating periodic tasks in both intra- and inter-ECU scenarios.

We consider two types of determinism: **(i) time determinism**, which guarantees predictable timing for sending and receiving operations, and **(ii) data-flow determinism**, which ensures that a consumer job (i.e., periodic task instance) always reads the output value from the same producer job, but does not ensure predictable timing for communication operations. Data-flow determinism is hence a weaker property than time determinism. We always consider partitioned fixed-priority scheduling, typically used in automotive, in which tasks cannot migrate and are statically assigned to cores.

The proposed model includes three variants: *High-Priority* (**HP**), *Timestamp* (**TM**) and *Mixed* (**TM-HP**). The **HP** variant leverages high-priority tasks to manage intra- and inter-ECU communication operations, ensuring both time and data-flow determinism by enforcing communication operations in correspondence of the task periods, inspired by approaches for (non-SL) LET for multi-core platforms [10]. Communication is managed using shared memory variables handled by tasks, one for each core, running at the highest priority, which are not subject to scheduling delays because they cannot be preempted. These tasks can be modeled as Generalized Multi-frame Tasks [10], [13] since they are activated in correspondence with the period of communicating tasks, as mandated by LET to ensure time-deterministic communication. Differently, the **TM** variant employs timestamps embedded into the message structure to enforce data-flow determinism, reducing the overhead caused by high-priority tasks but sacrificing strict time determinism. The timestamp field contains the time from which the message becomes valid for the producer, computed as the release time of the producer job, adding the task period and the communication/synchronization delay. Input and output communications are directly managed by application tasks at task periods, resulting in network operations being executed with the same priority as the application tasks. With **TM**, all the communications are managed via network using the SOME/IP protocol, maintaining a paradigm as close as possible to the current Adaptive Platform standard. However, the network stack gets involved every time a send or receive operation is performed, even in the case of intra-ECU communication, resulting in additional overhead. For this reason, the variant **TM-HP** has been designed to leverage both features of **HP** and **TM** variants. Network-based inter-ECU send and receive operations are handled directly from the application tasks, as in the **TM** variant. However, intra-ECU communications are managed by high-priority tasks with the aid of shared memory, removing the need to involve the network stack and enforcing time-determinism.

B. Modeling the extensions

To integrate SL-LET into the Adaptive standard, several key challenges have to be addressed:

CH1 Design a mechanism to perform input/output operations synchronized with the tasks' periods.

CH2 Abstract the SL-LET logic from the application level, allowing to maintain the same programming paradigm (i.e., keep the same signature for most of the APIs).

Figure 2 depicts the UML class diagram of the proposed extensions to the AUTOSAR Adaptive standard, enhancing the model with new classes embedding the SL-LET logic. The extension focuses mainly on the Event objects, as the approach can be replicated easily also for Methods and Fields.

AUTOSAR Standard Although most of the software running on automotive ECUs is designed in a time-driven fashion (i.e., based on periodic tasks), AUTOSAR Adaptive's service-oriented communication triggers tasks following a message-driven activation pattern. However, the standard also supports periodic execution of tasks as documented in the Timing Extension Specifications [14]. We therefore needed to properly "*connect*" these two parts of the standard to allow periodic communication.

In Figure 2, blue classes represent already existing classes that support periodic activations of tasks, which are essential to integrate the SL-LET logic. In particular, it is possible to specify the period, or inter-arrival time, of time events, which can activate specific software components. Note that despite some classes already referring to the SL-LET paradigm, they refer only to the periodic behavior, as the SL-LET logic is still missing in the standard.

Model Extension The core of the proposed model extension is based on three main classes whose purpose is to add support for the SL-LET paradigm at the application programming level. To represent tasks that follow the SL-LET paradigm, the novel `SLLETExecutable` class has been introduced. The `TDEventSLLET` class, which represents the timed event of the SL-LET software component, defines the task routine's periodicity. In compliance with the SL-LET protocol outlined in III-A, the `SLLETExecutable` class has been designed to provide complete flexibility in selecting the implementation strategy. The behavior of the APIs exposed by the class is strictly dependent on the specific implementation. With the **HP** variant, the `SLLETExecutable` class is designed to implement the high-priority tasks that handle the communication operations on behalf of the application tasks. The `Register()` and `Deregister()` APIs are used by application tasks to register and deregister themselves to the SL-LET paradigm, respectively. Note that the registration phase is **mandatory** to instruct the high-priority task of the presence of a new task for which to perform communication operations. For the **TM-HP** variant, the behavior of the class is the same as explained in the previous case, with the only difference it only affects intra-ECU communication. Finally, with the **TM** variant, this class refers to an adaptive executable, ignoring the behavior of the public APIs `Register` and `Deregister`.

To ensure GIOTTO's semantics with the high-priority task strategy in the intra-ECU communication phases, the `SLLETSynch` class has been created. This semantics specifically mandates that all output operations must be completed before moving on to the input ones when several tasks call for communication activities to be completed simultaneously. To maintain causality in a task chain, GIOTTO semantics is required. The `SLLETSynch` class lists all the tasks that synchronize during the data input/output phase, as shown in the model in Figure 2.

Figure 3 shows the behavior of the `SLLETSynch` class with three high-priority tasks performing LET operations on different cores. Each task must register to the write/read phase using the functions call `RTW` and `RTR` as shown in Figure 3, which correspond to the APIs of the `SLLETSynch` class `RegisterToWrite` and `RegisterToRead`, respectively. A spin-wait phase is undertaken to actively wait for all the other high-priority tasks (scheduled in other cores) to terminate the write phase and register to the read phase. Note that registering a task for the read phase implies deregistering it from the write phase. A synchronization barrier, which high-priority activities may subscribe to through the APIs depicted in Figure 2, can be used to construct such a class at the implementation level. `SLLETSynch` has been designed as a singleton-type class, allowing only one instance of the class to manage the synchronization of high-priority tasks running across the platform's various cores. This class is not used under the **TM** variant, as any type of communication, including intra-ECU, is managed via network. Lastly, in order to properly manage the SL-LET logic, the `SLLETTimingManager` class was designed to expose the temporal activations of the SL-LET SWCs to the classes associated with communication events. Specifically, SWC SL-LETs can use the `SetActivationTime()` API to record their activation time at each execution iteration. The `GetActivationTime()` API, on the other hand, can be used to appropriately handle communication events and make them run at the start of the task's actual activation.

Auto-generated Classes Similarly to AUTOSAR Classic, the Adaptive standard also heavily relies on automatic code generation. To tackle the challenges **CH1** and **CH2**, the full SL-LET logic has been confined within the specified classes. The proposed extension provides additional auto-generated classes for both sending and receiving events, directly derived from the event classes already existing in the standard (`EventDispatcher` and `Event`). This approach, in fact, allows us to minimize the impact on the current specifications, requiring less effort for the integration, and it is hopefully beneficial towards standardization efforts we want to pursue in future work. The SL-LET model is the same regardless of the specific implementation chosen.

C. *Configuration files*

Although all the auto-generated classes shown in Figure 2 are independent of the configuration of the system and agnostic of the possible protocol's approaches, it is still required to specify the periodicity of the process and to designate all the events that are managed under the SL-LET logic. Configuration can be done by modifying ARXML files already existent in the Adaptive Platform standard.

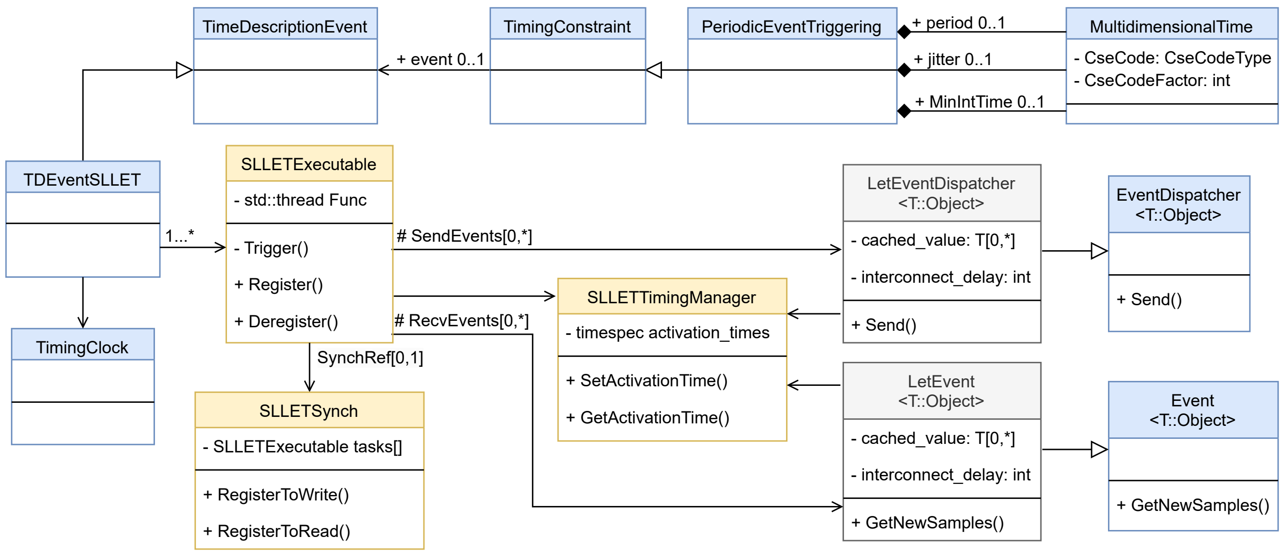

Fig. 2. Proposed extensions to AUTOSAR Adaptive. Classes in blue are already present in the Adaptive standard. Classes in yellow are added to the proposed extension. Classes in gray are autogenerated.

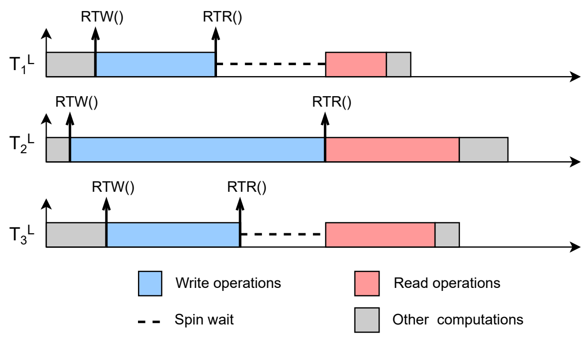

Fig. 3. GIOTTO semantic behavior enforced by `SLLETSynch` class. The `RWD` and `RTR` calls refer to `RegisterToWrite` and `RegisterToRead` APIs respectively.

TABLE I
SL-LET Communication Services Declaration.

```
<PROVIDED-SOMEIP-EVENT-GROUP>
  <SHORT-NAME>ProvEvents</SHORT-NAME>
  <EVENT-GROUP-REF>apd/.../Grp1</EVENT-GROUP-REF>
  <SL-LET>true</SL-LET>
</PROVIDED-SOMEIP-EVENT-GROUP>
                ...
<REQUIRED-SOMEIP-EVENT-GROUP>
  <SHORT-NAME>ReqEvents</SHORT-NAME>
  <EVENT-GROUP-REF>apd/.../Grp2</EVENT-GROUP-REF>
  <SL-LET>true</SL-LET>
</REQUIRED-SOMEIP-EVENT-GROUP>
```

Table I shows a part of the `Service Instance Manifest` ARXML file that is used to configure all the communication services exploited by a process. The `PROVIDED-SOMEIP-EVENT-GROUP` and `REQUIRED-SOMEIP-EVENT-GROUP` sections of the configuration file refer to all the provided and required group of events of the software component that will be used by the process. For each of these sections, it is required to specify the name of the group of events, along with its reference created in a separate interface ARXML configuration file and the flag to specify whether the group of events must be managed with the SL-LET logic. If the `SL-LET` flag is disabled, all the events belonging to the group will be handled according to the Adaptive Platform standard. As shown in Table I, for the provided events group `ProvEvents`, an instance of the `LetEventDispatcher` class shown in Figure 2 (which embeds the SL-LET logic) is created for each event of the group. This instance will be referred to by the software component every time a new message needs to be published. Similarly, for the required event `ReqEvents`, an instance of the `LetEvent` class is created and will be referred to by the software component every time a new message must be received The instances of the `LetEventDispatcher` and `LetEvent` classes, which are part of the proposed model extension, are auto-generated and linked to the process referred by the ARXML file when building the system.

Table II shows a part of the `Process Design` ARXML file that is used to configure the process. The `TD-EVENT-SLLET` section has been added to specify the *Time Description Event* used to determine the periodicity of a SL-LET executable, as highlighted in Figure 2. It contains the name of the event and the reference to the adaptive executable, which will adhere to the SL-LET paradigm, according to Section III-B. The section `PERIODIC-EVENT-TRIGGERING` specifies the parameters required to trigger a designated event periodically. According to the Generic Structure Documen-

TABLE II
SL-LET PROCESSES CONFIGURATION.

```
<TD-EVENT-SLLET>
  <SHORT-NAME>Activation</SHORT-NAME>
  <EXECUTABLE-REF>apd/.../Process1</EXECUTABLE-REF>
</TD-EVENT-SLLET>
              ...
<PERIODIC-EVENT-TRIGGERING>
  <SHORT-NAME>ReqEvents</SHORT-NAME>
  <EVENT-REF>apd/.../Activation</EVENT-REF>
  <PERIOD>
    <CSE-CODE>3</CSE-CODE>
    <CSE-CODE-FACTOR>100</CSE-CODE-FACTOR>
  </PERIOD>
</PERIODIC-EVENT-TRIGGERING>
```

tation [15], the *CSE-CODE* parameter is used to specify the granularity of the period, while the *CSE-CODE-FACTOR* is the integer representing the period itself. In the example shown in Table II, during the building phase a trigger with a period of 100ms is created and linked to an instance of the TDEventSLLET class, with an identification name of *Activation*, and referring to the *Process1* adaptive executable. As discussed in Section III-A, the high-priority tasks used in the **HP** and **TM-HP** variants do not rely on a specific period, but on the arrival times of communicating tasks. In this case, the high-priority task shares the periodic triggers related to the processes it manages, as it is possible to refer to different *SLLETExecutable* instances with a single *TDEventSLLET*.

IV. AUTO-GENERATED CLASSES IMPLEMENTATION

This section briefly explains the implementation of the SL-LET logic embedded in the auto-generated classes shown in Figure 2. It is important to remark that the automatic code generation process is not being affected by the ARXML configuration file explained in Section III-C as the SL-LET logic embedded in the event objects is agnostic of the implementation strategy chosen.

LetEventDispatcher This class has been derived from the class EventDispatcher provided by the AP standard and extended to implement the output operations compliant with the SL-LET semantics (i.e., send events). The cached_value attribute is used to store the latest message requested to be sent. Every time a publisher requests to send a new message, the attribute value is overwritten. The interconnect_task_delay attribute refers to the synchronization delay between the local timezone and the master timezone, from which is derived the global time. The value of the attribute is considered meaningful only in the context of local timezones. Otherwise, if the send operation is performed in the context of the master timezone, the attribute will be ignored as the synchronization delay is zero. The function Send() has been extended to fill the message timestamp according to the SL-LET semantics. The timestamp value is computed starting from the activation time of the task's job that is calling the send operation, which is stored by the SLLETTimingManager class, as discussed in Section III-B.

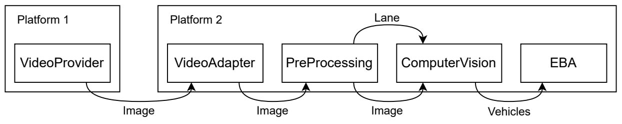

Fig. 4. Architecture of the Brake Assistant in APD.

Finally, the interconnect_task_delay value is added to the activation time retrieved before.

LetEvent This class has been derived from the class Event provided by the AP standard, and extended to implement the input operations (i.e., receiving operations) compliant with the SL-LET semantics. Similarly to the LetEventDispatcher class, the interconnect_task_delay attribute stores the timezone's synchronization delay. The cached_value list stores all the messages received from the network that are related to the event. The list is ordered based on the timestamp of the message received, adjusted with the local synchronization delay. The GetNewSamples() function has been extended to return the latest message received from the network whose validity time has expired and hence can be processed by the subscriber. All the messages with validity time not yet expired are stored in the cached_value list. The validity of the message is evaluated by comparing the message timestamp (adjusted with the local synchronization delay) with the activation time of the task instance that is requesting to receive new messages. Similar to the case of the send operation, the activation time is retrieved with the aid of the SLLETTimingManager class. If the adjusted message timestamp is antecedent to the activation time, then the validity is considered expired and the message can be delivered to the subscriber. Otherwise, the validity is considered not yet expired and the message must be stored in the cached_value list. Every time the GetNewSamples() function is invoked, first the cached_value list is inspected to find messages with validity time expired. If any, the most updated message is selected as the one to be processed by the subscriber, while the others are discarded. Then, the new messages received from the network are processed and stored in the queue of the Proxy class, if any. The function must return only the most updated messages between the one selected from the cached_value list and the latest message valid received from the network.

V. EVALUATION

The evaluation consists of an experimental setup based on a realistic automotive application, the Brake Assistant, made available by the AUTOSAR Consortium through the official Adaptive Platform Demonstrator (APD). As shown in [16], this Brake Assistant is a particularly interesting case since it highlights the lack of determinism in Adaptive and how a safety-critical application could be negatively impacted.

As shown in Figure 4, the Brake Assistant application is composed of five tasks distributed in two different platforms. The *VideoProvider* task is responsible for sending on the network the frames taken from the camera, with a period

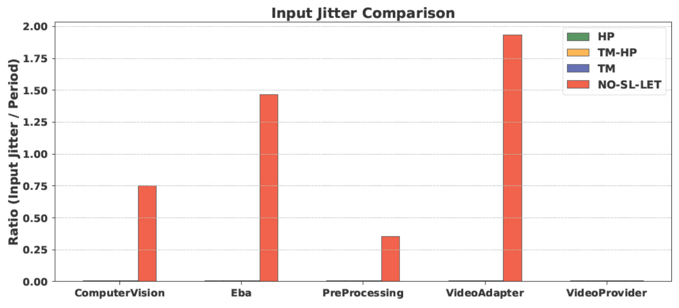

Fig. 5. Input jitter for different implementation approaches.

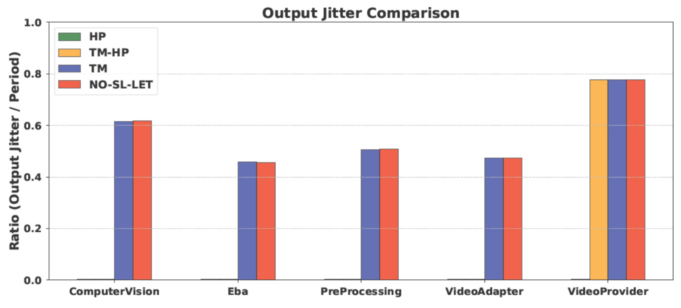

Fig. 6. Output jitter for different implementation approaches.

of 50ms. The *VideoAdapter* task periodically checks for new messages from the network containing frames sent from the *VideoProvider* task, with a rate of 25ms. The *PreProcessing* task receives the image from the *VideoAdapter*, recognizes the lane, and sends its bounding box, along with the full image, to the *ComputerVision* task, which will send the list of any detected vehicles in the lane to the *Emergency Brake Assistant (EBA)* task. Both *PreProcessing* and *ComputerVision* tasks have a period of 50ms, while the *EBA* task has a period of 25ms. For the evaluation, the Brake Assistant application has been split among two Raspberry Pi4 platforms powered by a quad-core 64-bit ARM SoC and running the Linux operating system. The communication is based on the SOME/IP protocol. Each task of the Brake Assistant application is designed as an AUTOSAR Adaptive software component, implemented as a Linux process, and scheduled by the fixed-priority scheduler of Linux (SCHED_FIFO). Each task of the Brake Assistant application executes a certain number of jobs in such a way that the total amount of messages exchanged by the whole chain of tasks is 10.000.

The purpose of the evaluation is to measure the average input and output jitters to check whether the three SL-LET variants have the expected behavior in terms of determinism. To this end, we define the output jitter of the k-th job of the i-th task τ_i as $J_k^o = |t_{(k+1)} - t_{s,k}|$, where $t_{(k+1)}$ is the activation time of the (k+1)-th job of the producer task and $t_{s,k}$ is the time in which the actual send operation is performed by the k-th job. The input jitter, instead, is $J_k^i = |t_r - t_a|$ where t_r is defined as the time after which the message can be read by the job k-th of the consumer to preserve the data-flow determinism, while t_a is the time the message is available to the consumer to be read. Figures 5 and 6 show the measured input jitter and the output jitters, respectively, comparing the variants described in Section III-A with the configuration without the SL-LET logic. These comparisons are made using the ratio input and output jitter, obtained by dividing the average jitter from all the jobs of each task with its period. The evaluation shows that HP removes both input and output jitter as proof that it guarantees time determinism. TM-HP and TM, instead, remove only the input jitter while still exhibiting output jitter because they can only guarantee data-flow determinism. With TM-HP, the output jitter is present only for the *VideoProvider* task, as it is the only one performing output operations by sending messages on the network (the others perform output using high-priority tasks). Fig. 5 also shows that the *VideoProvider* task has no input jitter with all the variants, as it is the head of the chain and, hence, its input is immediately available for all its jobs.

VI. Related work

Concerning the deterministic execution on POSIX OSs, the Linux community has designed and implemented the SCHED_DEADLINE scheduler [17], where tasks can be assigned a period and a runtime budget. To enforce periodic execution, the task can invoke the `sched_yield()` syscall to inform the scheduler of the end of the current instance; the scheduler will then wake up the task at the start of the next period. SCHED_DEADLINE is Linux-specific and, therefore, not available on other POSIX OSs. Moreover, the actual execution of the (input/output) tasks still depends on other high-priority SCHED_DEADLINE tasks in the system and can exhibit jitter. Several works considered protocols for LET in single-ECU platforms [10]–[12], [18]. Closer is the work in [19], which implemented LET on POSIX OSs but without considering AUTOSAR Adaptive. It relied on a bridge component (external to Adaptive) to artificially delay input/output operations. Differently, this paper shows how Adaptive can be extended to include native support SL-LET.

VII. Conclusions

This paper presented a model extension of the AUTOSAR Adaptive standard to integrate the SL-LET paradigm, which aims to lay the foundation for future standardization. An evaluation of the input and output jitter on a realistic automotive application showed that the model is general enough to accommodate different implementation strategies, which can guarantee different levels of determinism. Future work will investigate the standardization of SL-LET according to the approach proposed in this paper and evaluate the overheads of different implementation options.

References

[1] OSEK, “OSEK/VDX Operating System Specification 2.2.3,” https://www.osek-vdx.org/index_htm.html, Standard, Feb. 2005.
[2] AUTOSAR, “The AUTOSAR Classic Platform,” https://www.autosar.org/standards/classic-platform.

[3] ——, "Explanation of Adaptive Platform Design," https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_EXP_PlatformDesign.pdf.

[4] 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. 1–27, 2021.

[5] AUTOSAR, "Specification of Timing Extensions for Classic Platform," https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_TPS_TimingExtensions.pdf.

[6] "IEEE Standard for Information Technology—Standardized Application Environment Profile (AEP)—POSIX® Realtime and Embedded Application Support, Std 1003.13," 2003.

[7] AUTOSAR Consortium *et al.*, "SOME/IP Service Discovery Protocol Specification, Tech. Rep. 802," Tech. Rep., 2021.

[8] C. Scordino, A. G. Mariño, and F. Fons, "Hardware acceleration of data distribution service (DDS) for automotive communication and computing," *IEEE Access*, vol. 10, pp. 109 626–109 651, 2022.

[9] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree, "From control models to real-time code using Giotto," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 50–64, 2003.

[10] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 240–250.

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

[12] M. Beckert, M. Möstl, and R. Ernst, "Zero-time communication for automotive multi-core systems under spp scheduling," in *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–9.

[13] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, pp. 5–22, 1999.

[14] AUTOSAR, "Requirements on Timing Extensions," https://www.autosar.org/fileadmin/standards/R23-11/FO/AUTOSAR_RS_TimingExtensions.pdf.

[15] ——, "Generic Structure Template," https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_TPS_GenericStructureTemplate.pdf.

[16] C. Menard, A. Goens, M. Lohstroh, and J. Castrillon, "Achieving determinism in adaptive AUTOSAR," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 822–827.

[17] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[18] P. Pazzaglia, D. Casini, A. Biondi, and M. D. Natale, "Optimizing inter-core communications under the let paradigm using dma engines," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 127–139, 2023.

[19] D. Bellassai, A. Biondi, A. Biasci, and B. Morelli, "Supporting logical execution time in multi-core posix systems," *Journal of Systems Architecture*, vol. 144, p. 102987, 2023.