Automatic Deployment of Embedded Real-time Software Systems to Hypervisor-managed Platforms

Florian Schade*, Tobias Dörr*, Alexander Ahlbrecht†, Vincent Janson†, Umut Durak†, Juergen Becker*
*Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Email: {florian.schade, tobias.doerr, juergen.becker}@kit.edu
†German Aerospace Center (DLR), Institute of Flight Systems, Braunschweig, Germany
Email: {alexander.ahlbrecht, vincent.janson, umut.durak}@dlr.de

***Abstract*—The deterministic integration of concurrent functions on shared multicore platforms is a challenging yet important task. Especially in safety-critical environments, hypervisors can be used to achieve time and space partitioning, but their sole application is often insufficient to guarantee deterministic timing and data flow behavior. Considering the growing complexity of modern embedded systems, for example in terms of functionality and mixed-criticality requirements, model-based approaches are a promising starting point to tackle this issue. In this work, we bridge the gap between a model-based behavior specification methodology based on the Logical Execution Time (LET) concept and target platforms running a commercially available bare-metal hypervisor. Therefore, this paper describes a runtime environment that implements LET semantics at the level of hypervisor partitions and a tool-supported design methodology that deploys software to this runtime environment. From a behavior specification provided as a system model with annotated C code, the presented deployment tool generates binary images with guaranteed timing and data-flow behavior for the XtratuM hypervisor. The approach is finally validated by applying it to a Flight Assistance System (FAS) from the avionics domain.**

***Index Terms*—Model-based design, real-time systems, embedded software, Logical Execution Time (LET), hypervisors.**

I. Introduction

Modern embedded systems are often characterized by a high degree of concurrency. The operation of autonomous aerial vehicles, for example, is based on the continuous interaction of numerous sensing, computation, and actuation processes. Especially in safety-critical environments, this concurrency must often be implemented in a deterministic manner.

At the same time, there is a trend to integrate functions with different criticality levels on shared hardware platforms, for example in future avionics architectures [1]. To address the resulting mixed-criticality requirements, bare-metal hypervisors are important building blocks [2], since they provide time and space partitioning of shared hardware platforms. Since the sole application of a hypervisor is insufficient to guarantee deterministic timing and data flow behavior, achieving deterministic concurrency on such platforms remains a time-consuming and error-prone task. Model-based design methodologies that automate the deployment process can help to achieve a correct implementation in a timely manner.

This work targets the time- and value-predictable deployment of software components to hardware platforms running a bare-metal hypervisor. Therefore, we build up on the model-based behavior specification methodology from [3], which uses the Logical Execution Time (LET) paradigm [4] to achieve a platform-independent description of timing properties in concurrent software systems. Based on a timing-aware model of the envisaged software architecture and a code-driven programming model, the methodology from [3] performs deterministic model-in-the-loop simulations of specified system behavior. However, its focus lies on the specification and the simulation aspect. It comes without a runtime environment and provides no automatic deployment support. The goal of this paper is to close this gap.

More specifically, the contributions of this work can be summarized as follows:

1) An extension of both the software architecture metamodel and the code-driven programming model from [3] to capture deployment-specific inputs.
2) A runtime environment implementing LET semantics at the level of time-triggered hypervisor partitions.
3) A tool-supported strategy to deploy model-based behavior specifications to the proposed runtime environment.

The remainder of this paper is structured as follows: After a survey of related work in Section II, a high-level overview of the proposed design methodology is presented in Section III, which also covers the underlying metamodel. Section IV covers both the developed runtime environment and the tool-supported deployment strategy, before Section V presents the application of the presented approach in the form of a case study. Section VI closes the paper with conclusions and remarks on future research directions.

II. Background and Related Work

The LET paradigm was introduced in the time-triggered language Giotto [5], where functions are modeled in the form of periodically executed tasks. The behavior of each task is given as a sequential program that has the opportunity to read from the input ports and write to the output ports of its task. Logically, such reads and writes are required to take place at exactly the start and the end of a period, respectively, while the program's data processing logic can be executed anytime in between. This property decouples the real-time behavior of Giotto programs from a particular hardware platform or simultaneously executed software workloads [4].

Listing 1
HOOKS FROM THE ORIGINAL SWC API

```
void swc_init(void) {
    // SWC initialization logic
}

void swc_trigger(activation_id activation, const swc_port_map *port) {
    // SWC activation logic
}
```

Runtime systems such as the E machine [6] allow for a time- and value-predictable execution of Giotto programs.

The aforementioned concept to associate each task with a logical duration (from reading input ports to writing output ports) forms the basis of many other LET manifestations. Examples of such approaches are xGiotto for event-driven systems [7], the Timing Definition Language (TDL), which provides an improved syntax and simplified mode switching semantics [8], or the system-level LET concept [9], which is specifically optimized for distributed applications.

To achieve deterministic concurrency, especially under hard real-time requirements, various programming models apart from LET have been proposed. Examples include synchronous languages such as Lustre [10] and Blech [11], timed multitasking [12], or the reactor concept described in [13]. Such approaches are more expressive than LET-based concepts, but their compilation can be associated with additional challenges. Due to the underlying zero-delay abstraction of synchronous languages, for example, compiling them to target platforms can be complicated by causality issues [14].

Therefore, the LET concept is increasingly applied to the design of embedded systems. It has been integrated into the timing extensions of the AUTOSAR software architecture [15] and deployed to time-predictable multicore processors such as the Patmos [16] architecture, for instance.

III. MODEL-DRIVEN DEVELOPMENT METHODOLOGY

The entry point into the proposed development methodology is a model-based behavior specification. The approach from [3], which forms the basis of this work, expects the developer to describe the software architecture as a network of software components (SWCs), which are periodically triggered and communicate according to the LET paradigm. It further presents a programming model that allows developers to specify the functional behavior of a SWC using sequential code. The interface developers use to do so in a particular programming language is referred to as the *SWC API*; it was initially provided in the C programming language. As a sample excerpt from this API, Listing 1 shows two hooks that the developer is expected to populate with initialization and periodically triggered C code, respectively.

For the purposes of this paper, the methodology from [3] was selected due to a combination of properties that facilitate a time- and value-predictable hypervisor deployment. Most importantly, its usage of LET provides a deterministic abstraction from the target runtime, its channel-based communication model is compatible with industrial bare-metal hypervisors such as XtratuM [17], and its simulation framework can be used for verification and validation activities.

A. Key Design Decisions

The proposed methodology makes use of these properties to support the requirements of inherently concurrent systems, particularly with mixed-criticality and real-time requirements. This is reflected in the following key design decisions:

- **SWCs run on dedicated hypervisor partitions.** As introduced in Section I, sufficient isolation between software components needs to be ensured to prevent fault propagation among components of mixed criticality. Embedded hypervisors provide a certain degree of isolation by partitioning processing resources among guest software and enforcing resource limits. Thus, the proposed development mechanism isolates SWCs by implementing a one-to-one mapping of SWCs and hypervisor partitions, allowing for fine-grained resource access control.
- **The LET paradigm is applied on the level of SWCs.** To meet the aforementioned need for deterministic behavior of critical software in the context of a highly-concurrent system, a deployment runtime is used to enforce SWC execution and communication following the LET paradigm.
- **Environment interaction follows LET timing.** Interaction between the system under development and its environment shall be enabled by allowing SWC access to peripheral hardware components to control sensors, actuators, and external communication interfaces. To achieve a sufficient degree of determinism and thus maintain compatibility with simulation approaches such as [3], a runtime software is used to ensure that these interactions are in accordance with the specified LET timing.

B. Deployment Flow

The tool-supported strategy proposed in this paper is shown in Fig. 1. Its starting point is a system model ① created by the developer. The model has to be compliant with the extended software architecture metamodel, which will be presented in Section III-C. Thus, it specifies the system as a set of SWCs, defines communication interfaces between SWCs and towards the environment, and specifies LET parameters.

In addition to the system model, the developer needs to provide a functional implementation of each modeled SWC in the form of sequential code ②. The extended programming model (including the SWC API written in C) goes beyond the scope of [3] in the sense that it has full support for low-level driver accesses, e.g., to interact with sensors and actuators (see Section III-C and Section IV-D).

Using this system model, the deployment tool Ⓣ derives key deployment artifacts, such as the system schedule, memory map, and configuration parameters of the hypervisor partitions and deployment runtime. It then generates the corresponding configuration and build files. For each hypervisor partition, it creates a partition and runtime configuration ③ as well as a build configuration ④ in form of a Makefile that defines

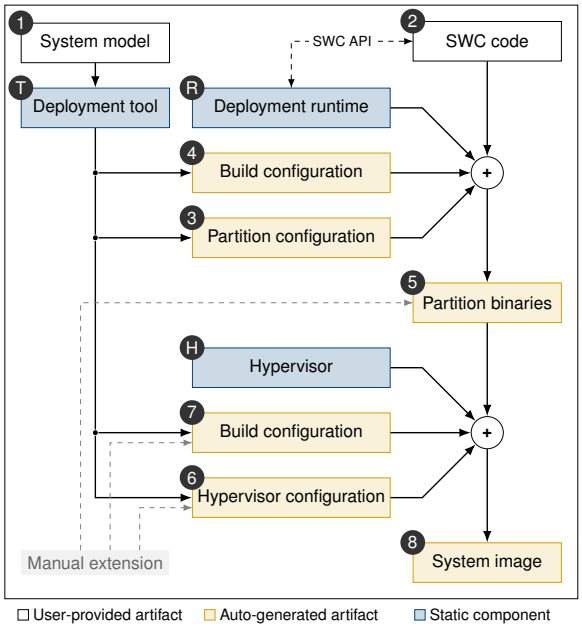

Fig. 1. Structure of the proposed deployment flow

the compilation process to generate the partition binary ⑤ from the SWC code, runtime configuration, and deployment runtime Ⓡ. In addition, it generates the hypervisor configuration ⑥ and the corresponding build configuration ⑦. Based on this configuration, all partition binaries are then linked with the hypervisor binaries and the hypervisor Ⓗ configuration as well as partition configurations to form the system image ⑧, which is ready for deployment on the embedded target platform.

If required by the system under development, the approach allows for the extension or modification of the generated artifacts at various points in the deployment flow. This can be helpful to include components that are not natively supported by the proposed deployment tool, e.g., to implement a SWC that wraps a Linux distribution. In this case, the developer has to consider the resulting implications on the system LET behavior and implement appropriate measures as required.

C. System Metamodel and Programming Model

The proposed deployment approach is based on a model that specifies key properties of the system under development. Figure 2 shows the corresponding metamodel, which is adopted from [3], extended to cover deployment information, and slightly modified regarding the modeling of the SWC execution timing. These changes, however, do not limit compatibility with simulation approaches as presented in [3].

The system architecture is modeled as a set of SWCs, each represented by a `SoftwareComponent` instance. SWCs represent sequential application code executed in a periodical pattern. The corresponding period is annotated as a property of the `SoftwareComponent` instance. The execution pattern within each period consists of one or more *activations*. An activation represents a single execution of the associated code and is modeled as an `Activation` instance.

To ensure deterministic timing and data flow, SWC code execution adheres to the LET paradigm. Thus, each activation is characterized by its logical runtime and its offset within the SWC period. These parameters define the *LET frame* of the activation, i.e., the time window within which the associated code is executed.

Inspired by the ARINC 653 standard [18], communication between SWCs is organized using ports and channels. A port (`SwcPort`) represents a communication interface of a SWC and implements a specific communication mode. Compatible ports can be connected via channels (`Channel`) and represent unidirectional data flow between two SWCs.

Two communication modes are supported: sampling and queuing behavior. Sampling ports allow for one-to-many communication. Thus, a sampling output port can be connected to multiple input ports via multiple channels. Sampling communication follows last-is-best semantics. Receivers will always read the latest value that was written by the sender. Senders can write values to these ports or perform a clear operation to indicate that no valid data is available. In contrast, queuing ports implement one-to-one communication following the first-in-first-out (FIFO) concept. Messages written to a queuing output port can be read at the corresponding input port in the same order. To bound memory usage, each queuing port has a defined capacity of messages that can be buffered.

Following the LET paradigm, data forwarding across ports happens in a time-triggered manner that is defined by the activation schedule. SWC code executed during an LET frame can access ports at any time via the SWC API. The underlying runtime has to ensure that inputs to the activation are read at the beginning of the LET frame and are not updated during the LET frame. Likewise, it needs to ensure that outputs written by SWC code do not come into effect until the end of the LET frame. Regarding writing operations to queuing ports, it is further defined that messages exceeding the port capacity are discarded. Likewise, when queuing channel inputs are read into an input port, messages exceeding the input port's capacity are discarded as well. This ensures that the channel buffer is cleared after reading by an activation, which simplifies the dimensioning of buffer capacities.

To further enhance the flexibility in regard to data flow specifications, port access can be defined on a per-activation basis using the `readFrom` and the `writeTo` property of an activation. Only ports explicitly referenced using these properties are updated during the corresponding activation.

Communication between SWCs and the system environment, such as sensors and actuators of the target embedded systems, is represented by ports as well. These ports have their `scope` attribute set to `EXTERNAL` and are referred to as *environment ports*. During deployment, the runtime forwards environment port data to target-specific low-level interfacing code as described in Section IV-D.

To cover deployment-specific information, we extended the original software architecture metamodel with a dedicated model annotation as highlighted in Fig. 2. It comprises a `DeploymentConfiguration` instance defin-

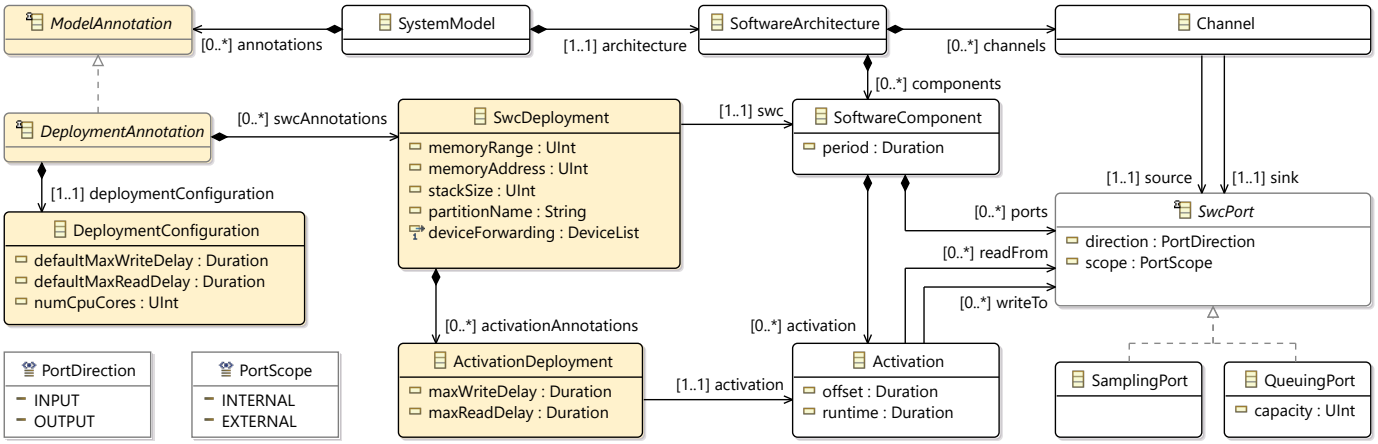

Fig. 2. Excerpt of the software architecture metamodel (with deployment-specific annotations in yellow)

ing target-specific information and one `SwcDeployment` instance per SWC to model SWC-specific information. Similarly, activation-specific information is modeled in one `ActivationDeployment` instance referring to the corresponding `Activation` instance.

IV. Deployment Mechanism

In this section, we present the proposed deployment mechanism, which can be structured into the three layers shown in Figure 3.

- The *runtime environment (RTE) layer* encompasses all management and OS-level software on the target platform, i.e., all software that is not part of the SWC code provided by the developer. This includes the hypervisor and the deployment runtime.
- The *tooling layer* encompasses the development tools that support the system developer to generate target configurations from the system model.
- The *modeling layer* encompasses the model-based description of the system. In the context of the deployment mechanism, it comprises the base metamodel presented in Section III-C as well as its extensions that enable the developer to annotate properties and constraints required by the deployment tool to generate the aforementioned configuration.

In the following, the central concepts of the proposed deployment mechanism are presented with respect to their manifestation in the three layers. These concepts are the mapping of SWCs to hypervisor partitions, the mapping of SWC activations to hypervisor time windows, and the implementation of LET-based communication using RTE-provided mechanisms. They are depicted in Fig. 3, references to this figure are denoted as [n].

A. Mapping of SWCs to Hypervisor Partitions

Embedded hypervisors provide a means to partition processing resources of computation platforms among guest software. This is achieved by creating logical partitions that contain the guest software. These partitions have limited access to

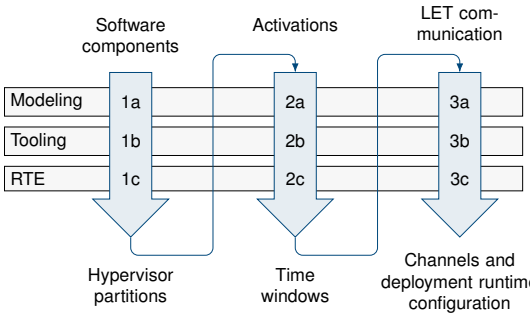

Fig. 3. Steps of the deployment mechanism

the platform's resources, such as CPU time, memory, and peripherals, and are connected via well-defined communication channels. Within a partition, hypervisors typically allow for the execution of both OS-level and application-level software. Thus, embedded hypervisors are a common component to support isolated execution of software on shared platforms. The proposed deployment mechanism makes use of this feature. Therefore, in step [1c], isolation between SWCs is achieved by deploying each SWC to a dedicated hypervisor partition.

The proposed deployment mechanism was implemented for the XtratuM [17] hypervisor. Its concepts, however, are transferable to other hypervisors that follow the concepts described in the ARINC 653 standard [18], such as PikeOS [19]. These hypervisors implement CPU sharing based on a fixed periodic schedule and provide channel-based inter-partition communication mechanisms that are compatible with the system metamodel from Figure 2.

To enable and control the execution of a SWC in a hypervisor partition, a deployment runtime is developed. The deployment runtime is system software that lies between the SWC code and the hypervisor. Based on the functionality provided by the underlying hypervisor, it implements the abstractions presented in Section III-C, such as LET-based SWC execution and inter-SWC communication.

To provide the necessary information for the configu-

ration of hypervisor partitions, the metamodel deployment annotation 1a comprises partition-specific information for each SWC. This includes the assigned memory range, partition name, and stack size of the hypervisor-provided in-partition RTE. In addition, peripherals that shall be accessible to a SWC partition are listed, such as *gpio* and *uart0*. This information is used by the deployment tool 1b during the generation of partition-specific configuration files. In doing so, the deployment tool resolves the device list entries into platform resource access configurations that are added to the hypervisor configuration.

B. Mapping of Activations to Time Windows

As described in Section III-C, SWC execution timing is specified by a periodic activation pattern. To implement this behavior on the target platform, we exploit the scheduling mechanism used on ARINC-653-oriented hypervisors. Here, schedules are defined in the form of a fixed-duration major frame (MAF), which is repeated cyclically. For each CPU core, the MAF defines a sequence of time windows, each of which has a fixed duration, an offset relative to the beginning of the MAF, and a reference to a partition. Thereby, it specifies the execution time window of the corresponding partition on the CPU core. Since SWC activation also follows fixed, periodic activation patterns, they can be mapped to time windows.

The system model 2a contains the required activation timing information in the form of LET parameters and the period of the corresponding SWC. To derive a hypervisor schedule, the number of available CPU cores needs to be specified as part of the `DeploymentConfiguration` instance, as depicted in Fig. 2.

Based on this information, the deployment tool 2b derives the hypervisor schedule in a three-step process. First, it determines the hyperperiod of all SWCs by calculating the least common multiple of the SWC periods. The hyperperiod equals the duration of the MAF. Secondly, it performs a periodic extension of all SWC activation patterns up to the determined hyperperiod by adding activations following the SWC's periodic activation pattern. Finally, it attempts to construct the hypervisor schedule by allocating one time window for each SWC activation, so that the following scheduling conditions are met:

- **S1** The time window offset and duration equal the corresponding activation offset and runtime.
- **S2** There is no overlap of time partitions mapped to the same CPU core.
- **S3** The maximum number of cores used equals the configured number of available cores.

The schedule generation mechanism processes SWC activations sequentially. For each activation, it attempts to add a corresponding time window to the schedule without violating the aforementioned constraints. In case that no solution is found, the process aborts and the developer is requested to adjust the SWC timing or assign more CPU cores.

At the RTE layer 2c, the resulting hypervisor schedule ensures that each SWC partition is executed during its time windows. The invocation of user-provided SWC code, however, needs to be controlled on a more fine-grained level and therefore is managed by the deployment runtime. The deployment runtime comprises an event-list-based scheduler to process actions within the SWC partition in a time-triggered way. The event list specifies events in combination with their temporal offset to the beginning of the MAF. Thus, the event list scheduler synchronizes with the hypervisor scheduler using hypervisor-issued interrupts at the beginning of the first time window assigned to an SWC in each MAF. The event list for each SWC is derived from the generated schedule by the deployment tool 2b and contains one activation event at the beginning of each time window, which leads to the execution of the `swc_trigger` hook (cf. Listing 1).

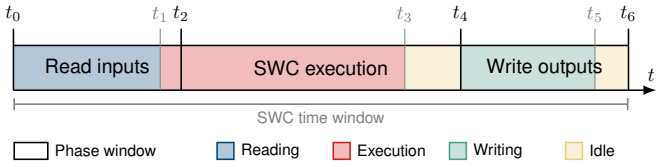

Fig. 4. Planned and sample on-target timing of activation execution phases

C. Enforcing LET Communication

The system model defines communication channels for inter-SWC communication, which either following sampling or queuing semantics. On the target platform, these channels are mapped to communication mechanisms provided by the hypervisor. ARINC-653-oriented hypervisors provide both queuing- and sampling communication channels between partitions, which are widely compatible with the mechanisms defined in Section III-C. Thus, modeled sampling and queuing ports can be mapped to their hypervisor-provided equivalent with few adaptions concerning port functionality.

Realizing LET-based communication on the deployment target poses a more complex challenge. As described in [4], LET program execution is correct "if the program reads input, in zero time, then executes, and finally writes output, again in zero time, exactly when the LET has elapsed since reading input." Thus, there are two main requirements in regard to the correct execution of an SWC activation: (I) Inputs to the SWC activation shall be read at the beginning of the activation's LET frame. When new input data becomes available after the beginning of an activation's LET frame, it shall not be visible to the running activation until the start of the next execution. Similarly, outputs shall come into effect not before the activation's logical runtime has elapsed. (II) Input reading and output writing shall be logically instantaneous.

The proposed deployment mechanism fulfills (I) by buffering port data between the SWC code and the hypervisor channels. This buffering mechanism is implemented as part of the RTE 3c. The buffer itself is provided by a target-independent part of the deployment runtime and is interfaced to hypervisor channels by a target-specific part. As depicted in Fig. 4, each time window representing an LET frame is split

into three phases: At the beginning of the time window, inputs to the SWC partition are read into local port buffers. During the subsequent execution phase, the SWC code is executed by calling the `swc_trigger` function. During this phase, port access is available via the SWC API provided by the runtime, resulting in read and write accesses to the local port buffer. At the end of the time window, outputs are forwarded from the port buffers to the SWC partition's output ports and thus written to hypervisor channels. To adhere to the programming model, the deployment runtime only reads inputs and writes outputs that are specified in the access lists of the activation and handles messages exceeding buffer capacities as described in Section III-C.

As communication across hypervisor channels cannot be done in zero time, (II) is not by itself fulfilled in real-world systems. To circumvent this issue and retain the guarantees given by the LET concept, the deployment mechanism requires the specification of the maximum executing times for each activation's input reading and output writing phase. Based on this information, it prevents the generation of schedules in which overlapping input reading and output writing phases can lead to nondeterministic system behavior. This timing information can be annotated in the system model 3a by defining an `ActivationDeployment` instance that references the corresponding activation. The `maxReadDelay` attribute represents the maximum duration of the input reading phase (t_0 to t_2 in Fig. 4). Analogously, `maxWriteDelay` represents the maximum duration of the output writing phase, i.e., t_4 to t_6 in Fig. 4. It is the responsibility of the developer to provide correct parameters, e.g., by conducting a worst-case execution time (WCET) analysis of the corresponding input reading and output writing logic as well as the activation logic.

Note that in practice this will lead to an overestimation of each phase's execution time, resulting in idle phases during the execution of an activation on the target platform (cf. Fig. 4). This is considered acceptable given the determinism guarantees that are obtained by the proposed approach. While idle phases caused by the overestimation of execution time cannot be avoided in a static, pre-determined schedule, idle phases in the SWC execution phase that result from an LET that exceeds the activation logic's WCET could be used to schedule other SWCs in parallel on the same core. In this case, it needs to be guaranteed that the execution time assigned to each SWC execution section equals at least the SWC code WCET and that input and output phases can be scheduled precisely at the specified time. Note that due to the buffering of inputs and outputs, the SWC code execution can start as soon as all inputs have been read (at t_1) as indicated in Fig. 4.

In the RTE layer 3c, the aforementioned phase concept is implemented using the event-list-based scheduler introduced in Section IV-B. Two additional event types are defined that trigger the input reading and output writing logic, forwarding data between internal port buffers and the hypervisor channels. The input reading event is scheduled as the first event at the start of the LET frame, followed by the activation event. The output writing event is scheduled at the beginning of the output writing phase to ensure that all outputs are written if the output writing logic requires the maximum configured execution time. Consequently, the deployment tool 3b generates the respective events during the generation of the deployment runtime configuration.

To ensure deterministic data flow within the system, an additional scheduling constraint is defined and adhered to by the scheduling logic implemented in the deployment tool:

S4 For any two SWC activations a_s and a_r, among which there is a sender-receiver dependency, the output writing phase of the sender must not overlap the input reading phase of the receiver. A sender-receiver dependency is given if a_s writes to a channel that is read by a_r.

The necessity of this constraint becomes clear when considering that the data transfer between hypervisor channels and the local port buffers is not atomic. Thus, parallel accesses to the set of channels read/written by an SWC can lead to indeterministic behavior. At the same time, the exact instant at which this data transfer occurs is not known and is expected to vary between execution iterations, which motivates the definition of sufficiently long read and write intervals.

D. Enabling Environment Interaction

Conceptually, environment ports represent the interface between the software system and its environment. In real-world systems, this interface is implemented by peripheral devices integrated with the processing hardware that are used to control on-board and external components, such as sensors, actuators, and communication interfaces. Due to the variety of peripheral hardware and connected devices, the integration of the corresponding driver code is left to the developer.

To facilitate the use of ports to represent environment interaction, the developer is required to provide code that forwards data between port structures and the corresponding peripherals. To integrate this code, the SWC API is extended by additional entry points named *environment access functions*. At runtime, these functions are called by the deployment runtime during the input reading and output writing phases of activations that access the corresponding environment ports. For input ports, the access functions are called during the input reading phase after all internal ports have been read. For output ports, they are called at the beginning of the output writing phase, i.e., before writing internal ports. This approach increases the overall schedulability of the system, since the scheduling constraint S4 only applies to inter-SWC communication.

As an example, Listing 2 shows the definition of an environment input port to read a button state, Listing 3 presents the generated environment access function, manually populated by code reading the corresponding hardware register.

V. Application in an Avionics Use Case

In the following, a practical application from the avionics domain will be used to demonstrate and evaluate the proposed deployment mechanism.

Listing 2
SYSTEM MODEL INCLUDING ENVIRONMENT PORTS

```
swc_controller: {
    // SWC period, activations, system ports, ...
    env_ports: {
        button_in: { data_type: "u8", mode: "sampling", direction: "in" }
    }
}
```

Listing 3
EXEMPLARY ENVIRONMENT PORT ACCESS FUNCTION

```
void swc_env_access_button_in(const port_access_u8_s_out *port) {
    uint32_t button_state =
        ((*gpio_bank1_data_reg) >> PB1_BIT) & 1UL;
    port->write(button_state);
}
```

A. Case Study Overview and Context

The development of improved traffic avoidance functionality due to continuously growing air traffic is a topic of increasing importance in the aviation domain [20]. Its relevance is underlined by the recent efforts to introduce a new Airborne Collision Avoidance System (ACAS) [21]. Accordingly, the selected use case for this study is the development and deployment of a Collision Avoidance System (CAS). The machine-learning-based (ML-based) implementation introduced in [22] serves as baseline for the demonstration. For the practical integration of such a ML-based SWC into an avionics system, sufficient isolation from other SWCs will be required. Hence, a hypervisor-based implementation is considered a suitable mechanism for deploying the CAS on an avionics computing module.

For the purposes of this case study, we consider the simplified Flight Assistance System (FAS) architecture shown in Fig. 5. A *Sensor Unit* receives transponder information, including the position, heading, and speed of the own aircraft as well as nearby aircraft (intruders). It is connected to a *Flight Assistance Computer* (FAC), which hosts the CAS software system and is the target of the presented deployment approach. Its software architecture consists of three SWCs. A preprocessing SWC (*preproc*) interfaces the *Sensor Unit*, prepares the incoming data, and forwards it to the *cas* SWC via channels (*AircraftState* and *IntruderState*). The *cas* SWC evaluates the situation and provides a *TrafficAdvisory* to indicate the recommended pilot action. Due to the ML-based nature of the CAS algorithm, the generated *TrafficAdvisory* is then checked for consistency by the *postproc* SWC and finally forwarded to the *Pilot HMI* to notify the pilot. To ensure that a potential failure of the *preproc* or *cas* SWC does not affect this consistency check, each SWC is required to run on a dedicated hypervisor partition.

In a safety-critical avionics system, the adherence of the deployed software architecture to the specification is essential for certification [23]. A central aspect is that critical properties such as the specified timing requirements are met. For the system at hand, the timing of the software architecture is specified as follows, resulting in the schedule depicted in Fig. 6. Within an overall period of $200\,\mathrm{ms}$, each SWC is assigned a logical runtime of $90\,\mathrm{ms}$, including $5\,\mathrm{ms}$ input reading and output writing phases. The *preproc* and *postproc* components are scheduled at the start of the period on separate CPU cores, while the *cas* component is scheduled with a $100\,\mathrm{ms}$ offset on the first core.

B. Case Study Setup and Execution

In this case study, only the FAC shown in Fig. 5 will be considered. The case study's main goal is to deploy and test the specified software architecture on the embedded hardware and to evaluate the resulting system's consistency with the specification. As the target FAC platform, the Zynq-7000 SoC was selected, which is supported by the XtratuM hypervisor.

To obtain the required artifacts for deployment, the system was modeled in the JSON format that serves as an input to the deployment tool. Implementations of all SWCs were provided, adhering to the programming model and API described above. Based on these inputs, the deployment tool was executed to generate the deployment configuration and files. The fully automated generation process resulted in all artifacts that were required to compile and then deploy the elaborated software architecture to the embedded hardware.

To verify the correct timing of inter-partition communication, and thus the correct functionality of the LET implementation, the runtime was amended by a logging functionality that records timestamps when hypervisor ports are accessed. This mechanism was used to evaluate the accuracy of the runtime in meeting the specified read and write phases.

C. Case Study Results and Discussion

Given test input from a flight simulator, the execution of the generated deployment image on the target platform produced the expected advisories. The result of the timing evaluation is depicted in Fig. 6, where the execution of read and write events is shown in the form of red markers. Our measurement results show that all recorded port accesses are carried out within the specified read and write phases, confirming the correct timing of the LET implementation in this case study. Note that all recorded events lie within $434\,\mu\mathrm{s}$ to $763\,\mu\mathrm{s}$ from the start of the corresponding read/write phase, indicating that the specified read/write phase duration could be significantly reduced.

For safety-critical applications, compliance of the system with the specification is essential. Therefore, an automated generation of deployment artifacts is very valuable to reduce the development effort, avoid potential mistakes, and ensure consistency between the specification and the implementation. In addition, the model-based nature of the proposed deployment process is expected to be helpful in handling complex architectures. However, the current evaluation did not focus on complexity, but rather on the feasibility of the process.

VI. CONCLUSION

To integrate highly concurrent mixed-criticality systems on powerful processing platforms, we presented a model-driven

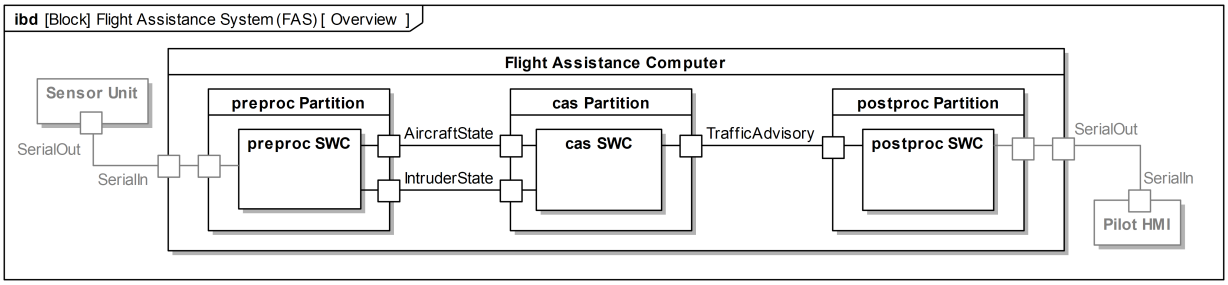

Fig. 5. Overview of the flight assistance system's architecture

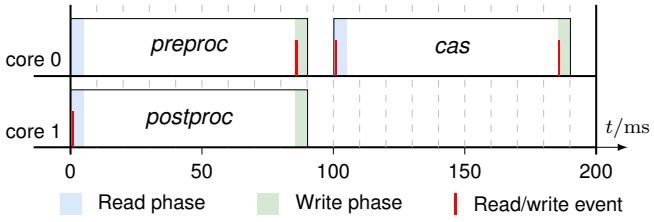

Fig. 6. FAC schedule and measured communication timing

mechanism that automates the generation of deployment artifacts and generates deployable system images. Our approach combines the Logical Execution Time (LET) paradigm with hypervisor technology to ensure deterministic timing and communication behavior on a system level as well as isolation between system elements on shared execution platforms. To achieve this, we presented an automated tool that derives configurations for the hypervisor and a hypervisor-based deployment LET runtime, along with the required metamodel for system specification. Promising directions for future work are the extension of the proposed mechanism to support distributed hardware architectures, tool support for SWC code that depends on complex operating systems, such as Linux, as well as optimized scheduling approaches to reduce idle periods and thus increase system efficiency.

Acknowledgment

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957210.

References

[1] T. Gaska, C. Watkin, and Y. Chen, "Integrated modular avionics - past, present, and future," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, 2015.

[2] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, "Virtualizing mixed-criticality systems: A survey on industrial trends and issues," *Future Generation Computer Systems*, vol. 129, pp. 315–330, 2022.

[3] T. Dörr, F. Schade, A. Ahlbrecht, W. Zaeske, L. Masing, U. Durak, and J. Becker, "A behavior specification and simulation methodology for embedded real-time software," in *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2022, pp. 151–159.

[4] C. M. Kirsch and A. Sokolova, *The Logical Execution Time Paradigm*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 103–120.

[5] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded Software*. Springer, Berlin, Heidelberg, 2001, pp. 166–184.

[6] T. A. Henzinger and C. M. Kirsch, "The embedded machine: Predictable, portable real-time code," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 6, Oct. 2007.

[7] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. A. Sanvido, "Event-driven programming with logical execution times," in *Hybrid Systems: Computation and Control*, R. Alur and G. J. Pappas, Eds. Springer, Berlin, Heidelberg, 2004.

[8] W. Pree and J. Templ, "Modeling with the Timing Definition Language (TDL)," in *Model-Driven Development of Reliable Automotive Services*, M. Broy, I. H. Krüger, and M. Meisinger, Eds. Springer, Berlin, Heidelberg, 2008.

[9] R. Ernst, L. Ahrendts, and K.-B. Gemlau, "System level LET: Mastering cause-effect chains in distributed systems," in *44th Annual Conference of the IEEE Industrial Electronics Society (IECON 2018)*, 2018.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[11] F. Gretz and F.-J. Grosch, "Blech, imperative synchronous programming!" in *Languages, Design Methods, and Tools for Electronic System Design*, T. J. Kazmierski, S. Steinhorst, and D. Große, Eds. Springer, Cham, 2020.

[12] J. Liu and E. Lee, "Timed multitasking for real-time embedded software," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 65–75, 2003.

[13] M. Lohstroh, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," in *Cyber Physical Systems: Model-Based Design*, R. Chamberlain, M. Edin Grimheden, and W. Taha, Eds. Springer, Cham, 2020.

[14] F. Siron, D. Potop-Butucaru, R. de Simone, D. Chabrol, and A. Methni, "The synchronous Logical Execution Time paradigm," in *ERTS 2022*, Toulouse, France, Jun. 2022, hal-03694950.

[15] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Trans. Cyber-Phys. Syst.*, vol. 5, no. 2, jan 2021.

[16] F. Kluge, M. Schoeberl, and T. Ungerer, "Support for the logical execution time model on a time-predictable multicore processor," *ACM SIGBED Review*, vol. 13, no. 4, Nov. 2016.

[17] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge, "XtratuM: a Hypervisor for Safety Critical Embedded Systems," in *Proceedings of the 11th Real-Time Linux Workshop*, 2009.

[18] ARINC, "ARINC specification 653 avionics application software standard interface part 1 – required services," 2010.

[19] SYSGO GmbH, "Avionics Solutions for Safe & Secure IMA and beyond," Tech. Rep.

[20] AIRBUS. (2022) Global services forecast (GSF) | 2022-2041.

[21] EUROCONTROL, "Airborne collision avoidance systems (ACAS) guide," Tech. Rep., 2022.

[22] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019.

[23] RTCA, "DO-178C - software considerations in airborne systems and equipment certification," 2011.