A Behavior Specification and Simulation Methodology for Embedded Real-Time Software

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

***Abstract*—Safety-critical real-time systems must be carefully designed to guarantee both functional and temporal correctness. State-of-the-art approaches to achieve this are often based on formal notations capturing both the desired functionality and relevant timing properties. This work is concerned with the design of embedded software systems for emerging fields such as the Urban Air Mobility (UAM) sector. In this context, it deals with scenarios that benefit from a less formal programming model, but for which guarantees on functional and timing behavior must still be provided. We propose a concept to specify and simulate the behavior of embedded real-time software in a deterministic manner. It combines the Logical Execution Time (LET) paradigm with a flexible, code-based approach for behavior specification and performs discrete-event (DE) simulations to determine how exactly the designed system responds to given stimuli. We describe this concept, present a reference implementation using Ptolemy II as simulation backend, and discuss its application to a pilot assistance system from the UAM sector.**

***Index Terms*—Model-based design, real-time systems, embedded software, Logical Execution Time (LET), Ptolemy II, discrete-event simulation, avionics.**

I. Introduction

Embedded systems in safety-critical environments are often required to guarantee functional and temporal correctness.

To facilitate the development of these systems, synchronous languages such as Lustre [1] or Esterel [2] have been proposed. They are based on a programming model where concurrently executed programs respond to inputs in an instantaneous manner [3]. Compiling synchronous programs to suitable target platforms leads to an implementation for which guarantees on functional and timing behavior can be derived. This zero-delay abstraction is also referred to as the Zero Execution Time (ZET) paradigm [4]. Today, synchronous languages are commonly used for the design of real-time systems in various domains, for example as part of the SCADE tool suite [3]. However, the zero-delay assumption can turn the compilation of a synchronous program into a challenging task [5]. In addition, from a usability point of view, designers in a particular use case might prefer to treat a system as interacting software entities with non-negligible execution times. If this is the case, an abstraction in which these times are explicitly specified can be beneficial.

The time-triggered programming language Giotto [6], which introduced the Logical Execution Time (LET) abstraction [4], replaces zero-delay with unit-delay computations. It considers periodically executed tasks, where each task reads its inputs at exactly the start of its period and writes its outputs at precisely the end of its period [6]. Deployed to a target platform that enforces this abstraction, a Giotto program exhibits deterministic timing and data-flow behavior. Since the period of a task serves as a time buffer that can be used for inter-task communication, the paradigm is particularly suited for distributed applications. From a software development perspective, it supports the integration of arbitrary code, e.g., written in the C programming language. Especially in the automotive domain, it is therefore increasingly used to achieve temporal correctness in on-chip and system-level scenarios. It is available, for instance, as part of the AUTOSAR timing extensions [7]. Compared to synchronous languages, however, the lack of a formal model that describes the behavior of individual tasks makes it difficult to reason about the functional correctness of LET-based systems.

The development of embedded software for emerging fields such as Urban Air Mobility (UAM) is increasingly subject to requirements not fully addressed by conventional design methodologies. Unmanned Aerial Vehicles (UAVs), for instance, are embedded into a regulatory environment where adherence to strict safety requirements is necessary [8]. At the same time, they require an integration of computationally intensive features such as collision avoidance or passenger entertainment systems [9]. Combined with the trend towards a consolidation of functions on multiprocessor system-on-chip (MPSoC) devices [10], this creates a need for approaches that support the explicit specification of non-negligible execution times, the efficient compilation to complex target platforms, and an automatic derivation of functional and temporal guarantees.

To address these challenges, we present a tool-supported behavior specification and simulation methodology based on the LET paradigm. It supports the integration of arbitrary code to describe the envisaged behavior of software entities and executes automatically generated discrete-event (DE) simulations to determine how exactly the overall system will respond to certain stimuli.

978-1-6654-9799-2/22/$31.00 ©2022 IEEE

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

- a **behavior specification methodology** that combines a metamodel capturing the software architecture of a system with a code-based programming model to describe the functional behavior of individual software entities;
- a **simulation strategy** that translates specified behavior into deterministic DE simulations and allows the user to execute these simulations in custom environments; and
- a **reference implementation** of the proposed concept that employs the Ptolemy II framework [11] as its backend to execute generated simulations.

The deployment of specified behavior to target platforms, which involves the actual scheduling of software entities, goes beyond the scope of this paper. Conceptually, this process is equivalent to the compilation of traditional LET programs [6], but special care must be taken to guarantee consistency with all details reflected in the DE simulation. The associated challenges will be tackled in future work.

This paper is structured as follows: We first give a high-level overview of the proposed design methodology in Section II. Section III describes the behavior specification step in more detail, while Section IV focuses on the simulation strategy. Our implementation is described in Section V, before Section VI covers related work and Section VII closes the paper with our conclusions and future research directions.

II. High-Level Concept

The starting point of the proposed design methodology is a platform-independent model of software entities and their interaction in the envisaged system. We refer to this model as the software architecture and to each entity it contains as a Software Component (SWC). Data flow from an output of a SWC to an input of a SWC is referred to as a channel. In the methodology, the LET paradigm is applied at this level of abstraction. More specifically, a *task* from the LET programming model [4] is conceptually equivalent to a SWC, while inter-task communication is represented via channels.

Furthermore, a SWC can be simulated as an *actor* as it is defined in the Ptolemy II context [11]. Using a pilot assistance system from the UAM sector, the next section illustrates how this capability contributes to the goal of this work.

A. Motivation: Simulation of a Pilot Assistance System

The system used as a motivational example is the Tactical Air Risk Mitigation System (TARMS) shown in Fig. 1.

It is equipped with a Terrain Awareness and Warning System (TAWS) that, according to DO-367 [12], has to issue warnings and cautions in response to situations such as an excessive rate of descent. In addition, the Collision Avoidance System (CAS) employs a neural network trained with flight maneuvers [13] to issue advisories for the prevention of collisions with intruder aircraft. The input and output ports of the TARMS embed it into the scope of the surrounding aircraft. More specifically, the Instruments block generates messages (z) to request certain actions from the aircraft's

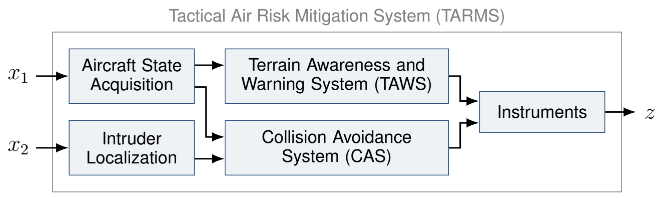
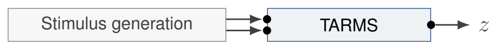
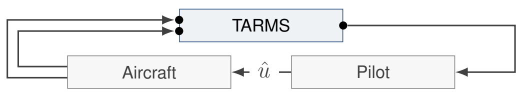

Tactical Air Risk Mitigation System (TARMS)
x_1
x_2
Aircraft State Acquisition
Intruder Localization
Terrain Awareness and Warning System (TAWS)
Collision Avoidance System (CAS)
Instruments
z

Fig. 1. Logical architecture of a sample system



(a) Open-loop case



(b) Closed-loop case

Fig. 2. Possible simulation setups integrating the TARMS

instruments. Analogously, the Aircraft State Acquisition and the Intruder Localization functions query the aircraft for its state (x_1) and information on surrounding vehicles (x_2), respectively. For the purposes of this paper, we map each logical function shown in Fig. 1 to a dedicated SWC. The ability to implement these SWCs using arbitrary code is desirable, since it allows us to make use of a readily available Rust [14] implementation of the TAWS and to realize the CAS by integrating C code that was automatically generated using a state-of-the-art machine learning library.

Formal methods to analyze safety properties of embedded systems, such as the approach presented in [15], can be applied to derive requirements that the TARMS needs to fulfill. An example of these requirements is the following: "When the rate of descent is at least (...) and the height above terrain is (...), then a Mode 1 caution alert is emitted within 2 seconds." Due to its formulation as an end-to-end requirement from x_1 to z, it cannot be met by a functionally correct implementation of the TAWS itself. The application of the LET paradigm supports the designer to achieve temporal correctness at the system level, but the nominal functionality of the Instruments block must still be considered: if it prioritizes its inputs in an unsuitable manner, the TAWS-related deadline might be missed. Therefore, the simulation shown in Fig. 2a is a simple yet powerful approach to collect evidences for the fulfillment of the above requirement.

In addition, this strategy makes it is possible to execute a model of the system in the context of its environment. From a control systems point of view, this environment usually captures the controlled plant and the plant's physical surroundings. In such a simulation, SWC actors are concurrently executed with actors that represent the heterogeneous environment. The closed-loop simulation in Fig. 2b shows a possible application of this capability. Using an executable model of the pilot's response to TARMS signals ($\hat{u}$) and a model of the aircraft itself (including its physical surroundings), it is possible to gather additional evidences for the verification of specified system behavior.

B. Logical Execution Time (LET) in the Context of SWCs

The original LET concept combines the time-triggered execution of tasks with support for mode switches [6]. More specifically, in GIOTTO, a set of *modes* serves as the top-level entity of a program. For each mode, the designer needs to specify periodic tasks as well as conditions that transition the program into another mode. From a modeling point of view, each task consists of input and output ports. Furthermore, entities referred to as *connections* represent the data flow between such ports [6]. From an implementation perspective, sequential code with known Worst-Case Execution Time (WCET) needs to be provided for each task. Tasks are periodically invoked. Logically, they have to read values from their input ports at exactly the start of their period and write values to their output ports at exactly the end of their period, both in zero time. During this period, the sequential code needs to be executed to produce the required output values. In other words, tasks exhibit *read-execute-write* semantics [16]. Finally, note that every port keeps its value until it is updated [17]. This means that inter-task communication exhibits last-is-best semantics or, in other words, that queuing is not directly supported.

Like an LET task, a SWC from our concept has input and output ports, is associated with user-provided code that is periodically executed in a time-triggered manner, and has a logical runtime that constraints port access as outlined above. Unlike conventional LET tasks, however, the notion of a SWC has been specifically designed to facilitate a deployment to targets such as a real-time operating system compliant with ARINC 653 [18] or the Classic Platform of AUTOSAR. Therefore, a port in our concept supports both sampling and queuing semantics. The logical runtime of a SWC is further decoupled from its period, which can lead to significantly reduced end-to-end latencies [19]. Finally, the concept defines a programming model that imposes specific requirements on how user-provided code needs to interact with input and output ports to facilitate a deterministic simulation.

The following definitions introduce the terminology we use with respect to LET-related aspects of the methodology.

Definition 1. The set of all SWCs in a software architecture is given by $S = \{s_i \mid 1 \leq i \leq N\}$. In what follows, a symbol with index i will always relate that symbol to s_i, while N denotes the number of SWCs in the system.

Definition 2. For every $s_i \in S$, an *LET frame* of s_i is a time interval that corresponds to exactly one execution of the code provided for this SWC; it is referred to as L_i.

Note that starting from time $t = 0$, a conceptually infinite sequence of L_i instances is associated with each $s_i \in S$.

Definition 3. The sequence of LET frames for $s_i \in S$ is completely specified by the *LET parameters* associated with the SWC. These parameters are given as $\ell_i = (r_i, \tau_i, T_i)$, where r_i denotes the logical runtime of the SWC, τ_i describes the time offset at which the first L_i begins, and T_i refers to the period between two consecutive releases of L_i.

A tuple of LET parameters ℓ_i is valid if and only if the conditions $0 < r_i \leq T_i$ and $0 \leq \tau_i < T_i$ are met.

Definition 4. The beginning of a specific L_i is referred to as *activation event* of s_i. Such events occur at the points in time given by $t = \tau_i + j \cdot T_i \ \forall j \in \mathbb{N}_0$.

Definition 5. The termination of a specific L_i is referred to as *termination event* of s_i. It occurs exactly r_i time units after the activation event that initiated the LET frame.

C. Overview of the Design Methodology

The proposed methodology in Fig. 3 can be divided into two portions: a *behavior specification* and a *simulation* step. Each step is associated with a tool that generates artifacts from provided user inputs as well as from previous generation results and built-in libraries, if applicable. They are referred to as the *GEN tool* and the *SIM tool*, respectively.

The entry point into the methodology is a software architecture model. Combined with metadata, it is captured in a user-provided input artifact we refer to as the *system model* (1). Invoking the GEN tool (2) on this artifact triggers an automatic generation of a *SWC code skeleton* for each specified SWC (3). Using an Application Programming Interface (API) we developed as part of this work, the toolchain user populates these skeletons with platform-independent *SWC code* (4). The result of this procedure is a deterministic description of the envisaged system-level behavior—both from a functional and from a timing point of view. Combined with a built-in library that implements the functions of the API, it can be passed to a backend that deploys it to a specific target platform.

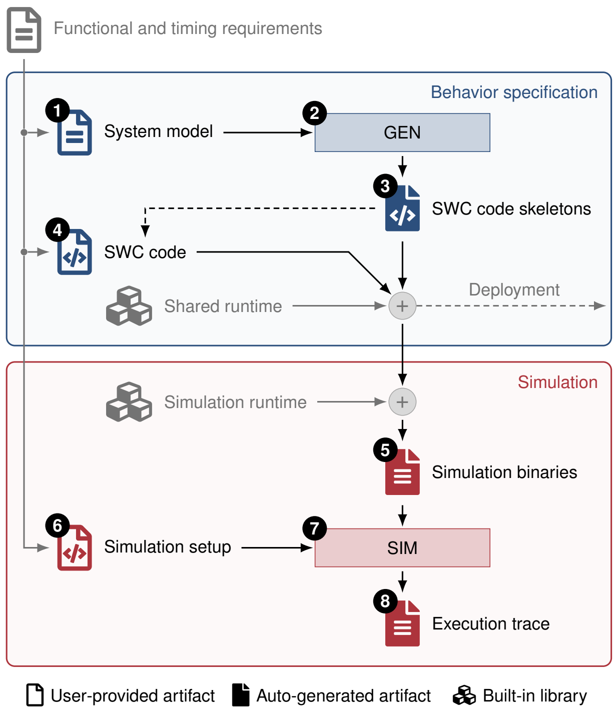


Fig. 3. Flowchart of the proposed design methodology

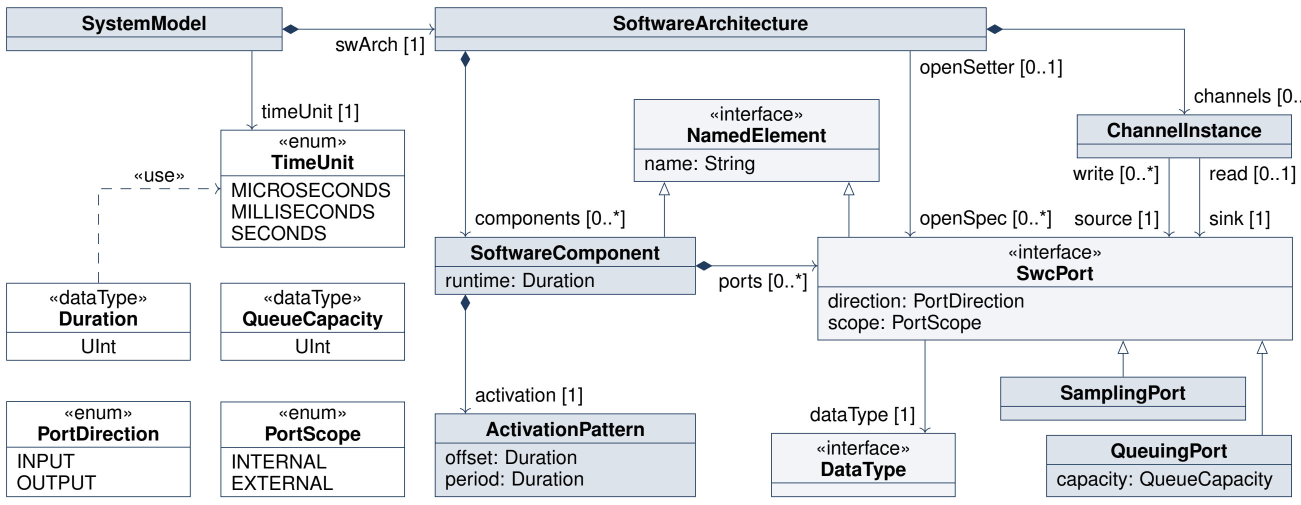


Fig. 4. Metamodel to describe input artifacts processed by the GEN tool

As described above, however, the deployment process is beyond the scope of this work. Therefore, the remainder of the development flow continues to consider logical time only. Synchronization with physical time, for example as part of the interaction with a sensor node on a target platform, is part of future work on the deployment concept.

Definition 6. The *specified behavior* of a system under consideration is its system model combined with user-provided code for each SWC in that model. It is both logically and temporally deterministic.

Given the specified behavior, a *simulation binary* is automatically created for each SWC (5). This artifact is an executable for the host platform that, whenever it is invoked, simulates an LET frame of the respective SWC by executing its associated code and responding with all outputs generated by the execution. Based on these binaries, the user composes a *simulation setup* (6), i.e., a textual description of how the specified behavior shall be executed. Most importantly, this description captures the environment to simulate, e.g., by instantiating a behavioral model composed using Ptolemy II. Invoking the SIM tool (7) on this description performs a DE simulation of the specified behavior in the respective environment by spawning and repeatedly invoking every simulation binary. While the simulation takes place, the tool generates an *execution trace* (8), which contains a deterministic description of all activation and termination events. Every such event is associated with all received input port values or, alternatively, all generated output port values.

III. Behavior Specification Methodology

In this section, the input artifacts that need to be provided to the behavior specification step of the methodology are described in more detail.

A. Modeling of the Software Architecture

The class diagram in Fig. 4 shows the metamodel that inputs provided to the GEN tool need to conform to. The entry point into an object hierarchy instantiated from this metamodel is a `SystemModel` object. It configures the base time unit used in the model and contains a `SoftwareArchitecture` instance. As already described in Section II, a software architecture consists of its SWCs and the channels that interconnect them. In the metamodel, this is reflected by the `SoftwareComponent` and `ChannelInstance` objects that are contained in a `SoftwareArchitecture`. The logical runtime r_i of an $s_i \in S$ is set directly in the object representing s_i. Its remaining LET parameters are captured in an `ActivationPattern` object, where the `offset` property corresponds to τ_i and the `period` property represents T_i. The following OCL [20] constraints formalize the already introduced validity requirements of these parameters:

```
context SoftwareComponent
inv: activation.period >= runtime
inv: runtime > 0

context ActivationPattern
inv: offset < period
```

Since all these properties are of the `Duration` type, i.e., they are represented as unsigned integers, the requirement that $\tau_i \geq 0$ for all $1 \leq i \leq N$ is met by definition.

Input and output ports are modeled as `SwcPort` instances contained in the `ports` property of the `SoftwareComponent` they are part of. For a queuing port, which is represented by the `QueuingPort` class, the `capacity` specifies the length of the queue. Queues must have room for at least one value. Therefore, the following invariant must hold:

```
context QueuingPort
inv: capacity > 0
```

Using the `SwcPort::direction` property, a port is set as either an input or an output. The `SwcPort::scope` property determines whether this port is used for communication within the system or for communication with the environment. An internal port (`scope = INTERNAL`) can only connect to other internal SWC ports. An external port (`scope = EXTERNAL`) must not be connected as part of the software architecture. Instead, in an actual deployment of the system, these ports implement interactions with external entities, e.g., using sensors or actuators.

Example. *Consider again the TARMS example from Fig. 1. Under the assumption that every logical function block from this figure is mapped to a dedicated SWC, the alerts issued by the TAWS need to be modeled as an internal port. x_1, x_2, and z need to be modeled as external ports.*

Finally, the `SwcPort::dataType` property associates every port with the data type of the values it is able to handle. The available data types cannot be specified as part of the system model. Instead, they are provided by the GEN tool[1]. Every data type has a predefined, fixed size, e.g., 8 bytes for an IEEE double-precision floating point number or $8n$ bytes for an n-element array of such numbers (with an arbitrary but fixed $n \in \mathbb{N}$). This size allows backends and the presented simulator to allocate sufficiently large buffers for ports.

A channel is represented by a `ChannelInstance` object. It establishes a unidirectional connection from its `source` port to its `sink` port. A channel can only be specified between two internal ports of the same mode (either sampling or queuing) and of the same data type. Furthermore, it must lead from an output to an input port:

```
context ChannelInstance
inv: source.oclType() = sink.oclType()
inv: source.dataType = sink.dataType
inv: source.scope = PortScope::INTERNAL
  and sink.scope = PortScope::INTERNAL
inv: source.direction = PortDirection::OUTPUT
  and sink.direction = PortDirection::INPUT
```

The `write` property of a `SwcPort` contains references to all the channels for which it serves as the source. Therefore, an output port can write an arbitrary number of channels. The `read` property of a `SwcPort` references the channel from which it receives its data. An input port can be associated with at most one channel. Internal input and output ports that are not connected to a channel must be explicitly specified as "open" using the `openSpec` property of the software architecture. This requirement acts as a sanity check for model instances and can be formalized as follows:

```
context SoftwareArchitecture
inv: components.ports
  ->select(scope = PortScope::INTERNAL)
  ->select(openSetter->isEmpty())
  ->forAll(read->notEmpty() or write->notEmpty())
inv: openSpec
  ->forAll(read->isEmpty() and write->isEmpty())
inv: openSpec
  ->forAll(scope = PortScope::INTERNAL)
```

[1]However, the design methodology is able to deal with any data type whose values can be stored using a statically allocated buffer. The specific GEN tool we introduce in Section V makes use of this property to allow custom data types to be defined using a flexible plugin mechanism.

The name of `SoftwareComponent` and `SwcPort` instances is subject to the following constraints:

```
context NamedElement
inv: name->notEmpty() implies name.size() > 0

context SoftwareArchitecture
inv: components->isUnique(name)

context SoftwareComponent
inv: ports->isUnique(name)
```

An instance of this metamodel, which is now completely described, can be supplied to the GEN tool. The next section presents the programming model that the user is expected to follow during the population of SWC code skeletons.

B. Development of Code for SWCs

The programming model we present as part of this work is defined as an API formulated in the C programming language. Although not covered by this work, the development of alternative APIs that provide a higher-level abstraction using languages such as C++ or Rust is conceivable.

Targeting this API, the GEN tool creates two hooks for each SWC specified in the software architecture:

```c
void swc_init(void);
void swc_invoke(struct swc_port_map *port);
```

The `swc_init` hook is intended for non-recurring initialization steps, while `swc_invoke` captures code to execute during an LET frame of the respective SWC. Code supplied to each of these hooks must be guaranteed to behave deterministically on all supported target platforms. Most importantly, this means that it must not exhibit undefined, unspecified, or implementation-defined behavior, while no functions with inherently non-deterministic behavior are called.

The `swc_port_map` passed to `swc_invoke` contains an accessor for each defined port. Depending on the data type and the mode (sampling or queuing) of the port, such an accessor allows clients of the API:

- to check the validity of a *sampling input* and, in case it is valid, to obtain the current value of the port;
- to determine if a *queuing input* is empty and, in case it is not, to obtain and remove the next value from the queue;
- to set the value of a *sampling output*; and
- to determine if a *queuing output* is full and, in case it is not, to push a value into the queue.

The exact nature of such an accessor is determined by the data type and, therefore, under the control of the GEN tool.

Example. *The aircraft state passed to the TAWS in the system from Fig. 1 consists of various state variables. For the purposes of this example, we consider a subset of these elements: the longitude, the latitude, and the altitude of the aircraft. The former two state variables are given by an angle encoded using a signed 32-bit variable, while the altitude is given by a length encoded as an unsigned 32-bit variable. The output provided to the Instruments block is a sink rate* *alert state encoded using an unsigned 8-bit variable. Modeling the inputs as sampling ports and the output as queue with a capacity of one leads to the following port map:*

```
struct swc_port_map {
  struct port_access_i32_s_in longitude;
  struct port_access_i32_s_in latitude;
  struct port_access_u32_s_in altitude;
  struct port_access_u8_q_out sink_rate;
};
```

Functions to interact with, e.g., a signed sampling port are wrapped in a `port_access_i32_s_in` *instance. Depending on how exactly the GEN tool implements the data type, a possible declaration of this data structure is as follows:*

```
struct port_access_i32_s_in {
  int32_t (*const read)(void);
  bool (*const is_valid)(void);
};
```

Using these interfaces, it is possible to access the input and output ports of the TAWS in a type-safe manner.

In the presented programming model, interactions with input and output ports have the following system-level effects:

- Writing a value to a *sampling output* buffers it within the SWC for the duration of the LET frame. At the termination event, the most recently written value is transferred to sampling inputs connected via a channel or, alternatively, released to the environment.
- Reading from a *sampling input* returns the value that the port held during the activation event of the LET frame. It retains this value until it receives a more recent one.
- Writing a value to a *queuing output* adds it to an internal first-in-first-out (FIFO) buffer of the SWC. At the termination event, all elements from this buffer are appended to the connected input queues or, alternatively, released to the environment. This clears the buffer.
- Reading from a *queuing input* returns the values that the queue contained during the activation event of the LET frame. More specifically, every read operation removes and returns the next FIFO value. Unconsumed values are discarded at a termination event.

The provided API allows developers of SWC code to query the status of individual ports and, therefore, to prevent illegal operations such as reading from an invalid sampling input or writing to a full output queue.

IV. Simulation Strategy

The specified behavior originating from the previous step is both logically and temporally deterministic (cf. Definition 6). The simulation strategy uses these properties to generate an executable DE simulation model of all SWCs along with their interactions via channel instances. To achieve this, each SWC is represented as an actor, while every channel instance is translated into a relation that forwards the outputs of an actor to the respective destinations. In addition, the strategy allows custom actors capturing the behavior of the environment

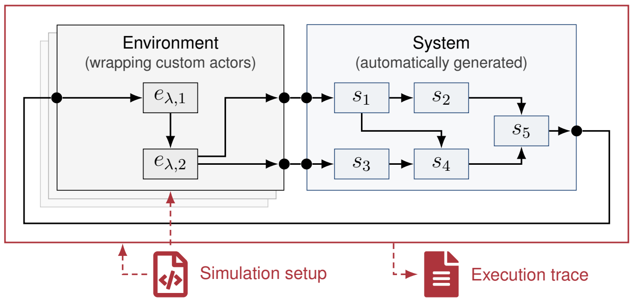


Fig. 5. Simulation strategy demonstrated using the TARMS example

to be instantiated into such a model. Therefore, it accepts user-generated behavioral models, interconnects them, and attaches them to environment ports of SWC actors.

The simulation setup shown in Fig. 3 defines exactly one such integration of environment actors into the DE simulation. In general, a user of the design methodology creates a set X of such setups, where a specific $\lambda \in X$ instantiates environment actors given by $E_\lambda = \{e_{\lambda,1}, e_{\lambda,2}, \ldots\}$. In the sense of a heterogeneous system simulation as it is supported, for instance, by Ptolemy II [21], it is possible to use a model of computation that differs from DE within an environment actor. A specific $e \in E_\lambda$ could, for example, be a continuous-time model. It is important to emphasize that non-deterministic behavior in environment actors (both from a functional and a temporal point of view) is generally permitted.

Example. *To simulate the closed-loop scenario introduced in Fig. 2b, a simulation setup ($\lambda \in X$) based on two environment actors can be created: $e_{\lambda,1}$, which captures how the pilot responds to TARMS signals, and $e_{\lambda,2}$, which represents how this response affects the aircraft state as well as the relative position to potential intruders. The internals of these actors (along with how their ports are connected) are specified by the creator of the simulation scenario. Based on this description, the SIM tool generates an executable model of the system, instantiates the two environment actors, and connects them as specified. This results in the auto-generated DE simulation setup visualized in Fig. 5.*

If at least one environment actor in a simulation setup exhibits non-deterministic behavior, repeated executions of this simulation can lead to traces that differ from each other. Although the timing and the sequence of LET frames captured in such traces will be the same in every case, the input and output values associated with simulated SWC events can vary. Such a simulation setup will generally have to be executed a large number of times to allow for a statistical analysis of resulting execution traces. While the proposed methodology is still applicable to such scenarios, fully deterministic simulation setups are of particular interest.

Definition 7. A *deterministic simulation setup* is a simulation setup in which every environment actor exhibits logically and temporally deterministic behavior.

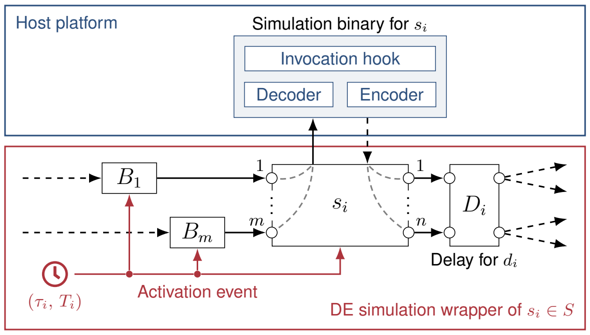


Fig. 6. Integration of a SWC into the DE simulation

Executing a deterministic simulation setup multiple times leads to identical execution traces. Therefore, only one execution of such a setup will generally have to be performed.

In the following section, we describe how to generate the executable simulation model in detail. Note that this procedure is identical for both deterministic and non-deterministic simulation setups.

A. Integration of the Specified Behavior

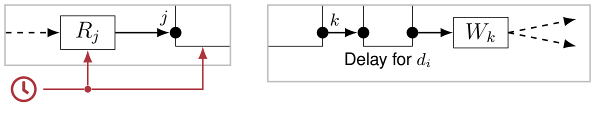
An executable model that captures the specified behavior of a specific SWC without its interactions with the environment is generated by instantiating the entities shown in Fig. 6.

In the DE simulation, the s_i actor represents the considered SWC itself. A discrete clock parameterized by τ_i and T_i is instantiated to generate a sequence of tokens, each of which representing an activation event of the SWC.

On the host platform (such as an x86-based desktop computer), the simulation binary associated with the SWC is spawned and connected to the DE simulation using a suitable inter-process communication mechanism. Via this connection, the SWC actor is able to trigger an LET frame execution, i.e., to call the invocation hook of the SWC by providing its simulation binary with input port values. In response, it will be provided with the outputs that the execution of the hook generates. Interactions with the simulation binary take zero model time, i.e., the s_i actor produces tokens that are synchronous with the activation event of the SWC.

Since interactions with the environment are currently neglected, only internal ports need to be considered. We refer to the number of such input and output ports as $m \in \mathbb{N}_0$ and $n \in \mathbb{N}_0$, respectively. Every input port $j = 1, \ldots, m$ is associated with an additional actor in the DE simulation: B_j, the *port buffer* managing that port. Analogously, all output ports $k = 1, \ldots, n$ are associated with a (conceptually combined) delay actor referred to as D_i. This actor is parameterized by d_i. It delays all tokens generated by the SWC actor to align them with the termination event.

Tokens traversing the edges in the DE simulation can be described as follows: The discrete clock generates a token whenever the global model time has advanced until an activation event of the SWC is due according to Definition 4. Such tokens are released solely to trigger the port buffers



(a) Input port (b) Output port

Fig. 7. Simulator integration of external ports

and the SWC; they do not carry a (meaningful) value. Edges directly or indirectly associated with a port carry tokens that capture input and output values of that port. More specifically:

- Edges associated with a *sampling port* carry tokens that represent exactly one such value.
- Edges associated with a *queuing port* carry tokens that encapsulate an ordered (FIFO) sequence of zero of more such values.

In any case, the simulator is unaware of the actual values that are wrapped within these tokens: the simulation binary makes use of the data type framework to provide the simulator with an encoded representation of every value, which is treated as a black box until it is decoded by a simulation binary.

In the top-level DE simulation, wrappers for all SWCs are generated as described above. Channel instances are then realized by directly connecting a dashed output arrow of the source port to the dashed input arrow of the sink port. This means that the port buffer B_j, which has not been described yet, logically represents a channel instance. Its purpose is to process the outputs generated by the source port in such a way that whenever it receives the activation event, it can provide the SWC actor with all the inputs that are necessary to implement the programming model described above:

- For a *sampling port*, it keeps track of the most recently received value and, whenever an activation event occurs, sends a token wrapping this value to the SWC actor.
- For a *queuing port*, it accumulates FIFO sequences contained in received tokens and, whenever an activation event occurs, sends the accumulated sequence to the SWC actor. This will clear the internal accumulator.

If a port buffer receives inputs simultaneously with an activation event, these inputs become available to the SWC actor as part of the immediately starting LET frame. Due to the fact that every input port is driven by no more than one channel instance (as indicated by the multiplicities in Fig. 4), no ambiguities between simultaneous events can arise.

B. Integration of Environment Actors

Environment actors are instantiated into the top-level DE simulation. To allow them to interact with external ports of SWCs, these ports are embedded into the simulation as illustrated in Fig. 7: an *environment reader* (R_j) is generated for each external input, while an *environment writer* (W_k) is generated for every external output of a SWC. R_j transforms tokens with native simulator data types into encoded values that the SWC actor can again forward as a black box. Furthermore, it samples the received value (for sampling ports) or accumulates received values (for queuing ports) in a manner similar to the port buffer. W_k reverses these actions. The dashed arrows shown in Fig. 7 are finally connected to environment actors as specified by the simulation setup. Note that the procedure to translate between encoded values and native simulator types needs to be explicitly implemented into the GEN tool for each available data type.

Definition 8. A data type has *environment support* if encoded values of that type can be created from and translated into values of a corresponding data type of the simulator.

A system model is invalid if a data type without environment support is specified for an external SWC port.

V. Implementation

The reference implementation we developed as part of this work targets the Java Virtual Machine (JVM). More specifically, both the GEN and the SIM tool are realized as command-line applications written in Kotlin. As the notation for system model instances, we created a human-readable syntax using the JSON5 format [22]. The backend used to perform DE simulations is Ptolemy II [11], which itself runs on the JVM. It is integrated into the SIM tool by linking against its libraries to generate DE simulations according to the rules given in Section IV. During the generation of the model portion that represents the specified behavior of the system itself, only a deterministic subset of the DE simulation capabilities provided by Ptolemy II is used.

The available implementation of the GEN tool provides common scalar data types (such as signed and unsigned integers) and allows toolchain users to implement their own types using a plugin system. Wherever it is possible, environment support is implemented for built-in types. The `i32` data type, for instance, which represents signed 32-bit integers, is mapped to the equivalent `int` data type of Ptolemy II.

To spawn simulation binaries, the SIM tool makes use of the `java.lang.Process` interface provided by the JVM. To establish bidirectional connections with these binaries, it launches a local TCP server to which they connect after executing their respective `swc_init` hook. These connections are used by SWC actors whenever a `swc_invoke` hook needs to be invoked to simulate an LET frame. Note that from a Ptolemy II perspective, the TCP-based execution of SWC actors violates strict actor semantics, i.e., the requirement that an actor must produce its outputs before changing its state [11]. However, the SWC actor integration strategy ensures that correct simulation results are still achieved.

To conclude this section, we give a brief description of how the reference implementation can be applied to a simplified version of the TARMS example.

Example. *We consider two SWCs from the TARMS context: one representing the TAWS and the other representing the Instruments block. The former has only one input: an external sampling port that reads the altitude in feet as a signed 32-bit integer from the environment. Its output is an internal queue with a length of one requesting the emission of TAWS alerts due to an excessive sink rate. The queue holds a '*`1`*' to request a caution alert and is read by the second SWC, which prioritizes all received inputs and, among other things, sets an external sampling output of Boolean type to* `true` *whenever a sink rate caution alert is emitted.*

To apply the toolchain to this example, we use the developed JSON5 notation to specify a software architecture with $S = \{s_1, s_2\}$, where s_1 represents the TAWS and s_2 represents the Instruments block. Ports are added as outlined above and, without loss of generality, LET parameters are set to $\ell_1 = (100\,\mathrm{ms},\, 0,\, 100\,\mathrm{ms})$ and $\ell_2 = (50\,\mathrm{ms},\, 0,\, 500\,\mathrm{ms})$.

Afterwards, the GEN tool is called, generated SWC code skeletons are populated, and the SIM tool is invoked to perform an open-loop simulation with an environment actor that generates a deterministic sequence of decreasing altitude values. At time $t_1 = 800\,\mathrm{ms}$, this actor provides the TARMS with an altitude of $1522\,\mathrm{ft}$, which necessitates the emission of a sink rate caution alert. Internally, the SIM tool translates this simulation setup into the Ptolemy II model shown in Fig. 8 and generates a textual trace that describes the observed sequence of activation and termination events. Captured events are displayed using ascending IDs. The following output is an excerpt of the generated trace in which selected lines have been manually removed for the sake of brevity:

```
15/instruments: TERMINATION at 550 ms
    - sink_rate_caution: S(false)
19/taws: ACTIVATION at 700 ms
    - altitude: S(1526)
21/taws: ACTIVATION at 800 ms
    - altitude: S(1522)
22/taws: TERMINATION at 900 ms
    - sink_rate: Q(1)
24/taws: TERMINATION at 1000 ms
    - sink_rate: Q(1)
25/instruments: ACTIVATION at 1000 ms
    - sink_rate_alert: Q(1, 1)
27/instruments: TERMINATION at 1050 ms
    - sink_rate_caution: S(true)
```

This excerpt shows that in the simulated case, a sink rate caution alert is emitted at time $t_2 = 1050\,\mathrm{ms}$.

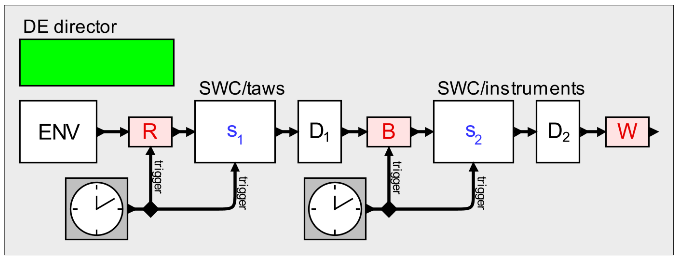


Fig. 8. Open-loop simulation synthesized in Ptolemy II

Combined with the guarantee that a correct deployment of specified behavior exhibits exactly the interactions visible in the trace, knowledge derived from such simulations serves as evidence for the fulfillment of functional and temporal requirements. This is especially the case if a deterministic simulation setup is used, i.e., if the behavior of the simulated environment is fully deterministic.

VI. Related Work

Our simulation strategy is similar to the approach presented in [23], which uses Ptolemy II to predict the runtime behavior of LET models. However, it is not associated with a programming model for the code-based behavior specification. The co-simulation concept proposed in [24] defines an interface between Ptolemy II and the HLA/CERTI framework. Therefore, it is a promising platform for an alternative implementation of the current simulation backend, which makes use of Ptolemy II alone. Evaluating the performance impact of such an implementation is an interesting path for future research.

The programming model PTIDES [25] defines a strategy for the deterministic execution of software in distributed systems. However, its focus lies on the synchronization between a physical and a logical timeline rather than the time-triggered execution of SWCs. It allows for a more flexible execution of software on target platforms but cannot be used to generate execution traces as they are derived by the SIM tool. The language Lingua Franca [26] has extensive support for the integration of user-provided code, but its PTIDES-based execution semantics differs again from the time-triggered nature of our methodology. The system-level LET concept [27] is a variant of the LET paradigm in which selected communication processes are explicitly modeled to reduce end-to-end latencies. Therefore, it is a promising idea for the extension of our current behavior specification methodology.

VII. Conclusion

To facilitate the development of systems whose requirements are poorly addressed by traditional design techniques, our concept combines the application of the LET paradigm with a novel, simulation-aware programming model. It supports the integration of arbitrary code and performs a deterministic simulation to derive guarantees about the runtime behavior of the system. Despite its flexibility from a programming point of view, we demonstrated that this approach qualifies as a structured framework to analyze the fulfillment of functional and timing requirements specified for modern real-time systems. Future work will focus on the application to more complex use cases, the scalability of the simulation strategy, and the deployment to target platforms.

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] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[2] F. Boussinot and R. de Simone, "The ESTEREL Language," *Proc. IEEE*, vol. 79, no. 9, pp. 1293–1304, Sep. 1991.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[4] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer, Eds. Springer, Berlin, Heidelberg, 2012.

[5] 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.

[6] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded Control Systems Development with Giotto," *SIGPLAN Not.*, vol. 36, no. 8, pp. 64–72, Aug. 2001.

[7] AUTOSAR, "Specification of Timing Extensions," AUTOSAR CP Release 4.4.0, Oct. 2018, ID: 411.

[8] A. P. Cohen, S. A. Shaheen, and E. M. Farrar, "Urban Air Mobility: History, Ecosystem, Market Potential, and Challenges," *IEEE Trans. Intell. Transport. Syst.*, vol. 22, no. 9, pp. 6074–6087, Sep. 2021.

[9] J. Athavale, A. Baldovin, S. Mo, and M. Paulitsch, "Chip-Level Considerations to Enable Dependability for eVTOL and Urban Air Mobility Systems," in *AIAA/IEEE 39th Digital Avionics Systems Conference (DASC '20)*, San Antonio, TX, USA, Oct. 2020.

[10] T. Gaska, C. Watkin, and Y. Chen, "Integrated Modular Avionics - Past, Present, and Future," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 30, no. 9, pp. 12–23, Sep. 2015.

[11] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

[12] RTCA, "DO-367: Minimum Operational Performance Standards (MOPS) for Terrain Awareness and Warning Systems (TAWS) Airborne Equipment," May 2017.

[13] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy Compression for Aircraft Collision Avoidance Systems," in *IEEE/AIAA 35th Digital Avionics Systems Conference (DASC 2016)*, Sacramento, CA, USA, Sep. 2016.

[14] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, and L. Ryzhyk, "System Programming in Rust: Beyond Safety," *Oper. Syst. Rev. (ACM)*, vol. 51, no. 1, pp. 94–99, Sep. 2017.

[15] A. Ahlbrecht and U. Durak, "Integrating Safety into MBSE Processes with Formal Methods," in *IEEE/AIAA 40th Digital Avionics Systems Conference (DASC '21)*, San Antonio, TX, USA, Oct. 2021.

[16] 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 '18)*, Washington, DC, USA, Oct. 2018, pp. 4084–4089.

[17] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming," in *Embedded Software (EMSOFT 2001)*, T. A. Henzinger and C. M. Kirsch, Eds. Springer, Berlin, Heidelberg, 2001.

[18] ARINC, "653P0-3 Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653," Nov. 2021.

[19] A. Naderlinger, "Subjecting Legacy Simulink Models to Timing Specifications," in *Cyber Physical Systems: Model-Based Design (CyPhy 2018)*, R. Chamberlain, W. Taha, and M. Törngren, Eds. Springer International Publishing, Cham, 2019.

[20] J. Cabot and M. Gogolla, "Object Constraint Language (OCL): A Definitive Guide," in *Formal Methods for Model-Driven Engineering (SFM 2012)*, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, Berlin, Heidelberg, 2012.

[21] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling Cyber-Physical Systems," *Proc. IEEE*, vol. 100, no. 1, pp. 13–28, Jan. 2012.

[22] A. Kishore and J. Tucker, "The JSON5 Data Interchange Format," Mar. 2018, version 1.0.0. [Online]. Available: https://spec.json5.org/

[23] P. Derler, A. Naderlinger, W. Pree, S. Resmerita, and J. Templ, "Simulation of LET models in Simulink and Ptolemy," in *Foundations of Computer Software: Future Trends and Techniques for Development (Monterey Workshop 2008)*, C. Choppy and O. Sokolsky, Eds. Springer, Berlin, Heidelberg, 2010.

[24] J. Cardoso and P. Siron, "Ptolemy-HLA: A Cyber-Physical System Distributed Simulation Framework," in *Principles of Modeling*, M. Lohstroh, P. Derler, and M. Sirjani, Eds. Springer, Cham, 2018.

[25] Y. Zhao, J. Liu, and E. A. Lee, "A Programming Model for Time-Synchronized Distributed Real-Time Systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, Bellevue, WA, USA, Apr. 2007, pp. 259–268.

[26] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a Lingua Franca for Deterministic Concurrent Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, Jul. 2021.

[27] 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, Apr. 2021.