International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/sxxxxx-022-xxxxx-x

MIMOS-Tools: an Integrated Toolchain for Model-Based Design of Real-Time Systems

Wang Yi · Chengzi Huang · Behnam Khodabandeloo · Duc Anh Nguyen

Received: date / Accepted: date
©

Abstract
Despite decades of progress in model-based techniques and tools for embedded real-time systems, there is still no integrated environment that supports the entire development lifecycle. This paper presents MIMOS-Tools, a comprehensive toolchain that spans all development phases, including modeling, simulation, formal verification, real-time scheduling, schedulability and end-to-end latency analysis, and code generation for heterogeneous multicore platforms. All phases are grounded in a single, coherent, and deterministic model of computation, ensuring that verified functional and timing properties are preserved throughout development and carried over to the final implementation. Unlike existing synchronous design tools such as Simulink and SCADE, MIMOS-Tools supports asynchronous communication and computation through FIFO queues and flexible real-time scheduling, rather than relying on offline static schedules typical of synchronous approaches. This enables modular and incremental design and implementation. At the same time, synchronous systems can be represented within the proposed asynchronous framework using periodic tasks with zero deadlines, in accordance with zero-time semantics (the synchronous hypothesis).
In this paper, we present the formal operational semantics of the underlying MIMOS model of computation—a multi-rate task graph model introduced in [48] for real-time system modeling—which forms the semantic foundation of MIMOS-Tools. We also provide a detailed account of the toolchain's key features and demonstrate its capabilities and industrial applicability through case studies across diverse application domains.
The toolchain and all models for case studies presented in this paper are currently available at https://github.com/uu-mimos/MIMOS-Tools.

Keywords Embedded Real-Time Systems · Dataflow Networks · Modeling · Simulation · Verification · Real-Time Scheduling and Code Generation

1 Introduction

Over the past decades, a broad range of theories, methods, and tools have been developed for the design, analysis, and deployment of embedded real-time systems. Major advances include modeling and simulation environments [43, 17], formal verification techniques e.g. [29], real-time scheduling and timing analysis [33, 9, 8, 45], time-triggered architectures [27, 21], and synchronous programming languages and industrial design tools [6, 19, 20, 16, 2]. These technologies have significantly improved the reliability and predictability of complex real-time systems and have been successfully adopted in industrial practice, including in safety-critical domains.

Despite this substantial progress, a complete and practical toolchain that supports the entire development lifecycle of real-time systems—from modeling and simulation to formal verification, real-time scheduling, system-level timing analysis, automatic code generation, and subsequent system evolution—remains unavailable. Existing solutions usually address only specific development stages and focus on particular classes of properties. For instance, formal verification often relies on abstractions that omit implementation details, creating a semantic gap between verified models and deployed systems. Conversely, real-time scheduling theory concentrates on non-functional aspects such as workload characterization and resource allocation while largely abstracting from functional behavior. As a consequence, engineers must manually integrate heterogeneous models and

Wang Yi, Chengzi Huang, Behnam Khodabandeloo, Duc Anh Nguyen
Department of Information Technology, Uppsala University, Uppsala, Sweden
E-mail: yi@it.uu.se
E-mail: chengzi.huang@it.uu.se
E-mail: behnam.khodabandeloo@it.uu.se
E-mail: ducanh.nguyen@it.uu.se

tools across multiple abstraction levels, a process that is error-prone and difficult to certify in safety-critical contexts.

The need for an integrated and semantically coherent development environment has long been recognized by both the Formal Methods and Real-Time Systems communities. Such an environment requires a model of computation that is expressive enough to capture functional and non-functional aspects while preserving a clear separation of concerns—for example, between functional correctness and timing correctness, and between computation and communication. It must also be deterministic so that transformations across abstraction levels preserve the intended behavior of the system under development and ensure that properties established early in the design flow remain valid after deployment. Strong composability is likewise essential to support modular development, incremental integration, dynamic updates, and long-term system evolution.

To address these requirements, the MIMOS model was introduced in previous work [48]. It is inspired by Kahn Process Networks (KPN) [24], a classical deterministic model for concurrent computation. While preserving functional determinism, MIMOS augments KPN processes with explicit timing and resource constraints. In this way, it defines both a *multi-rate task-graph model* and a real-time scheduling framework, enabling the enforcement of end-to-end timing requirements on the deterministic functional behavior of systems.

In MIMOS, systems are modeled as networks of periodic real-time tasks that communicate through asynchronous channels, allowing each task to execute at its own rate. This contrasts with the synchronous design paradigm, which relies on a global base clock and thus limits modularity and flexibility [1]. Unlike traditional Kahn process networks, which rely on blocking reads on FIFO queues, MIMOS adopts wait-free communication mechanisms: FIFO buffers with timed non-blocking reads and registers for sampled data. To ensure timing determinism, and inspired by the Logical Execution Time (LET) approach of Giotto [21], MIMOS restricts channel read and write operations to predefined time instants—specifically at task release and deadline—thereby preserving time-triggered task releases and communication independently of the actual scheduling of computations between these instants. This combination yields two fundamental properties of MIMOS. *Determinism* ensures that output streams are uniquely determined by input streams, enabling faithful system-level simulation, formal verification, and final implementation within a model-based development flow. *Composability* allows new components to be integrated without perturbing the behavior of existing ones throughout the development lifecycle.

[1] A detailed discussion of related work and comparisons is provided in Section 6.

In this paper, we present a formal operational semantics for the MIMOS model, which forms the foundation of MIMOS-Tools, a comprehensive end-to-end toolchain for real-time system development. By relying on a single, coherent, and deterministic model of computation throughout all stages, the toolchain eliminates semantic gaps between design and implementation and enables a consistent and reliable development process.

We also provide a detailed description of MIMOS-Tools's key features including:

- **Graphical modeling:** a visual, hierarchical environment with mixed-language support and LLM-assisted programming for rapid prototyping and iterative design.
- **Simulation and verification:** fast functional analysis and validation through simulation, complemented by formal verification to guarantee system properties.
- **Real-time scheduling and analysis:** task mapping from design models to real-time tasks; schedulability and end-to-end latency analysis using both analytical and simulation-based methods; and graphical simulation of run-time scheduling.
- **Target code generation:** a semantics-preserving translation from MIMOS models to executable multicore code whose execution retains the functional and timing properties verified in the earlier stages.

The paper is organized as follows: section 2 presents the MIMOS model, its operational semantics, and the determinism theorem. section 3 describes the design principles of the toolchain and its layered architecture. section 4 details the key features of MIMOS-Tools, illustrated with a drone controller example. section 5 demonstrates the toolchain's applicability through case studies. section 6 reviews related work and provides a comparison with MIMOS-Tools. section 7 concludes the paper.

2 The MIMOS Model: Multi-Rate Task Graph

We first clarify the terminology. By the MIMOS model (or simply MIMOS) we mean the formalism used to construct models, whereas a MIMOS model denotes a particular instance built in this formalism to represent or design a system.

MIMOS[2] is a formalism originally introduced in [48] for modeling, implementing, and updating real-time systems. Conceptually, it can be viewed as a *multi-rate task graph* model for real-time systems. In this section, we present an operational semantics for MIMOS to establish the formal foundation of MIMOS-Tools. This operational characterization provides a deterministic model of computation that is applied uniformly across all features of the toolchain.

[2] MIMOS stands for Multi-Inputs and Multi-Outputs Systems

We begin with an informal overview, followed by formal transition rules and the determinism theorem, which guarantees that systems specified in MIMOS behave deterministically with respect to both functionality and timing.

2.1 Informal Semantics

In MIMOS, a system is modeled as a directed graph where nodes represent computation *tasks* and edges represent communication *channels*. Tasks communicate exclusively via channels by exchanging *tokens* (values of specific data types).

Tasks and Channels. Each task may have multiple input and output channels and a *task function*, which defines a mapping from input tokens (read from input channels) to output tokens (written to output channels).

Tasks can be described using any real-time task models with deterministic release patterns as described in the survey [42]. For the current implementation, we support only periodic tasks (as other exisiting tools e.g. Simulink), characterized by a *period* (or time distance between consecutive releases) and a *relative deadline* (latest allowed completion time after release) [33]. Each task is also assigned a worst-case execution time (WCET), which defines the resource budget allocated to the task and is used for schedulability analysis and code generation[3]. This parameter does not affect the model's timing behavior, which depends solely on periods and deadlines, assuming tasks are schedulable under the given WCET bounds.

Channels are *single-writer and single-reader*: exactly one task writes to each channel and exactly one task reads from it[4]. Channels buffer *tokens* and are classified into two types:

- **FIFO queues** are unbounded first-in, first-out buffers that preserve the order of tokens. Each write appends a token to the tail, and each read removes and returns a token from the head if the FIFO is non-empty. A FIFO may be initialized with any finite number of tokens. Reads are assumed to be non-blocking and instantaneous, meaning the reader can immediately determine whether the FIFO is empty and access its head token if available.
- **Registers** store a single token. Each write overwrites the stored token, while each read returns it without removal. Registers are initialized with one token and therefore always contain a valid value. Reads from registers are instantaneous. Registers can also be used to maintain a task's local state by connecting them back to the task itself.

[3] We assume the WCET is validated by external tools, e.g., Absint's aiT [15].

[4] However, the contents of channels may be copied to multiple readers without changing the model.

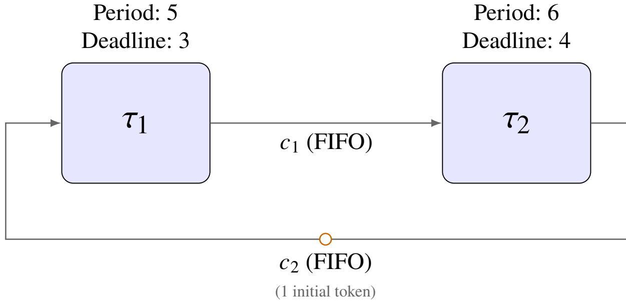
Task Release and Execution Pattern. Each task is released according to its assigned period. Every task release follows a three-phase *read–compute–write* pattern:

- **Read at release instant.** At its release time instant, the task performs an instantaneous, non-blocking read on all its input channels. If the task is *enabled* (i.e., all input channels are non-empty; this is always true for register channels, which always hold a valid token), it consumes one token from each FIFO input channel, reads the current value from each register input channel, and transitions to the *activated* state. Otherwise, the task *skips* this release: it consumes no tokens, produces no outputs, and its inputs are not checked again until the next release instant.
- **Compute within the period.** For an activated task, its task function is computed at some point between the release and the deadline. There are no constraints on when or how the computation occurs within this interval—it may execute immediately, be deferred, or even be distributed across multiple processors. It is assumed that the computation is fully scheduled and guaranteed to complete within the deadline by the scheduler[5].
- **Write at deadline.** At the deadline instant, the activated task writes the results computed by the task function to its output channels.

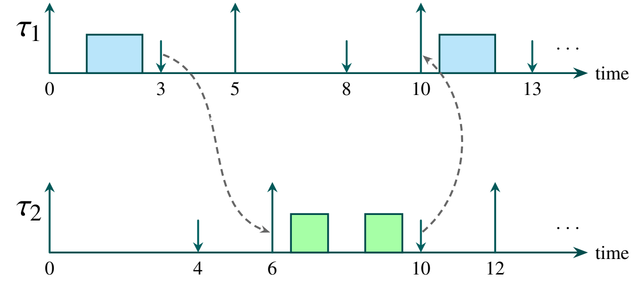
Note that the read and write operations of all tasks may occur only at deterministic time instants, predefined by each task's period and deadline. Furthermore, the results computed for writing to output channels must be ready between the scheduled read and write instants. When read and write operations occur at the same instant (possibly by different tasks), write operations are prioritized and executed before read operations (the *write-before-read* rule). This ensures that read operations always access the latest values written by preceding write operations, making the outcome of read and write operations at any instant deterministic. In other words, the contents of each channel are uniquely determined, since each channel has exactly one writer and one reader. Intuitively, this implies that the state representing the contents of all channels—starting from a given initial state—after any sequence of read, compute, and write operations over time is unique.

Figure 1 shows an example MIMOS model with two periodic tasks τ_1 (period 5, deadline 3) and τ_2 (period 6, deadline 4), connected by two FIFO channels: a FIFO c_1 from τ_1 to τ_2, initialized empty; and a FIFO c_2 forming a feedback loop from τ_2 to τ_1, initialized with one token. At time 0, both tasks are released: τ_1 reads the initial token from c_2 and ready to execute if scheduled, while τ_2 finds c_1 empty and skips this release. At time 3 (the deadline and pre-defined

[5] Completion without deadline violation must be verified a priori through off-line schedulability analysis.



(a) The two tasks with different periods and deadlines, communicating via FIFOs c_1 and c_2, with one initial token present in c_2.



(b) The instants for releases (up-arrows), deadlines (shorter down-arrows), computations (colored), communication (dashed).

Fig. 1 An example MIMOS model for a dataflow network and its execution timeline.

for write-operation), τ_1 writes to c_1. At time 5, τ_1 is released again but c_2 is empty, so τ_1 skips this release. At time 6, τ_2 is released again, consumes the token from c_1, and ready to execute if scheduled. At time 10, τ_2 writes to c_2 and τ_1 is simultaneously released; the *write-before-read* rule ensures τ_2's write completes first, so τ_1 successfully reads the new token and is ready to execute if scheduled.

2.2 Formal Semantics

This section formally defines the syntax of MIMOS and the behavior of MIMOS models as a labeled transition system and establishes the above informally stated deterministic property.

System Model and Configuration

A **System Model** in MIMOS consists of a finite set of tasks $\mathcal{T}$ and a finite set of channels C, forming a directed graph where nodes are tasks, edges are channels, and each channel is typed as either a FIFO or a register. Tasks are ranged over by τ and channels by c. Each task $\tau \in \mathcal{T}$ is defined by a tuple $(\mathsf{P}(\tau), \mathsf{D}(\tau), \mathrm{I}(\tau), \mathrm{O}(\tau), \mathsf{F}(\tau))$ where $\mathsf{P}(\tau), \mathsf{D}(\tau) \in \mathbb{N}_{>0}$ stand for the period and relative deadline[6], $\mathrm{I}(\tau), \mathrm{O}(\tau) \subseteq C$ denote the input and output channels, and $\mathsf{F}(\tau)$ specifies the task function. We assume that for any two distinct tasks τ, τ', $\mathrm{I}(\tau) \cap \mathrm{I}(\tau') = \emptyset$ and $\mathrm{O}(\tau) \cap \mathrm{O}(\tau') = \emptyset$ i.e., each channel has at most one reader and one writer.

Let $\mathcal{D}$ denote the domain of token values, where $\mathcal{D}^*$ is the set of all finite sequences over $\mathcal{D}$ and $\varepsilon \in \mathcal{D}^*$ represents the empty sequence. Since each task may have multiple input and output channels, we let v range over vectors of token values, with one component per channel. A scheduled write-operation is a tuple (τ, d, v), meaning that task τ writes v to its output channels at the future time d. A **System Configuration** is a tuple $(t, s, \mathsf{W}, \mathsf{R})$ where

- $t \in \mathbb{N}_{\geq 0}$ is the current time,
- $s : C \to \mathcal{D}^*$ is the state, mapping channels to their current contents[7].
- W is the set of write-operations scheduled to commit at future time instants and
- $\mathsf{R} \subseteq \mathcal{T}$ is the set of tasks released at t and ready to read their inputs.

We assume that the initial configuration of a system is $(0, s_0, \emptyset, \mathcal{T})$, where time starts at 0, the initial contents of channels are given by s_0, and $R_0 = \mathcal{T}$ (all tasks released at time 0).

Operational Semantics: Transition Rules

To formalize the transition rules, we define the precise meaning of channels read- and write-operations. The read operation $(\mathrm{v}, s') = r(\tau, s)$ reads the input channels of τ: for each $c \in \mathrm{I}(\tau)$, if c is a FIFO then $\mathrm{v}(c)$ is the head token of $s(c)$ and $s'(c)$ is $s(c)$ with its head removed, and if c is a register then $\mathrm{v}(c) = s'(c) = s(c)$. Channels not in $\mathrm{I}(\tau)$ remain unchanged. The write operation $s' = w(\tau, s, \mathrm{v})$ updates the output channels of τ: for each $c \in \mathrm{O}(\tau)$, if c is a FIFO then $\mathrm{v}(c)$ is appended to the end of $s(c)$, and if c is a register then $s'(c)$ is overwritten with $\mathrm{v}(c)$. Channels not in $\mathrm{O}(\tau)$ remain unchanged. We use $\mathsf{Released}(t) \triangleq \{\tau \in \mathcal{T} \mid \exists k \in \mathbb{N}.\ t = k \cdot \mathsf{P}(\tau)\}$ to stand for the set of tasks released at time t, and $\mathsf{W}(t) \triangleq \{(\tau, d, \mathrm{v}) \mid (\tau, d, \mathrm{v}) \in \mathsf{W} \wedge d = t\}$ for the set of write-operations scheduled for time t. Furthermore, we define the next event time $\mathsf{NextTime}(t) \triangleq \min\big(\{k \cdot \mathsf{P}(\tau) \mid \tau \in \mathcal{T},\ k \in \mathbb{N},\ k \cdot \mathsf{P}(\tau) > t\} \cup \{k \cdot \mathsf{P}(\tau) + \mathsf{D}(\tau) \mid \tau \in \mathcal{T},\ k \in \mathbb{N},\ k \cdot \mathsf{P}(\tau) + \mathsf{D}(\tau) > t\}\big)$, which is the earliest future time instant at which either a task is released or a write operation is scheduled. A task τ can compute if it has all inputs available, denoted $\mathsf{Enabled}(\tau, s)$ if $\forall c \in \mathrm{I}(\tau).\ s(c) \neq \varepsilon$.

The operational semantics is defined by three transition rules: (1) READ fires at time t when no write operations are scheduled at t, i.e. $\mathsf{W}(t) = \emptyset$, and there exists a released

[6] Note that both are non-zero for the natural reason to be feasible.

[7] Register channels store a single data value, viewed as a length-one sequence in $\mathcal{D}^*$.

task $\tau \in \mathsf{R}$ enabled in the current state s. The inputs of τ will be read, assigned to v, and will be used to compute the outputs $\mathsf{F}(\tau)(\mathrm{v})$ to write by the deadline of τ i.e. $t + \mathsf{D}(\tau)$. The precondition $\mathsf{W}(t) = \emptyset$ enforces the write-before-read rule: all writes scheduled at t must complete before any reads at that instant. (2) Write commits the write operations scheduled for time t one by one. (3) Next advances time when no write operations are pending and no released tasks in R are enabled. Time advances to $t' = \mathsf{NextTime}(t)$, and R is reset to the set of tasks released at t'. With these rules, we can establish the determinism theorem for MIMOS.

Transition Rules

Read

$$\frac{\mathsf{W}(t) = \emptyset \qquad \tau \in \mathsf{R} \qquad \mathsf{Enabled}(\tau, s) \qquad (\mathrm{v}, s') = r(\tau, s)}{(t, s, \mathsf{W}, \mathsf{R}) \longrightarrow (t, s', \mathsf{W} \cup \{(\tau,\ t + \mathsf{D}(\tau),\ \mathsf{F}(\tau)(\mathrm{v}))\}, \mathsf{R} \setminus \{\tau\})}$$

Write

$$\frac{(\tau, t, \mathrm{v}) \in \mathsf{W}(t) \qquad s' = w(\tau, s, \mathrm{v})}{(t, s, \mathsf{W}, \mathsf{R}) \longrightarrow (t, s', \mathsf{W} \setminus \{(\tau, t, \mathrm{v})\}, \mathsf{R})}$$

Next

$$\frac{\mathsf{W}(t) = \emptyset \qquad \forall \tau \in \mathsf{R}.\ \neg\mathsf{Enabled}(\tau, s) \qquad t' = \mathsf{NextTime}(t)}{(t, s, \mathsf{W}, \mathsf{R}) \longrightarrow (t', s, \mathsf{W}, \mathsf{Released}(t'))}$$

Theorem 1 (Determinism)
For any given sequence of transitions, the reached configuration starting from the initial configuration is unique.

Proof
By induction on the length of transition sequence. □

2.3 Further Extensions

The standard MIMOS model (formally defined in subsection 2.2) assumes that each task reads exactly one token from each input channel and writes exactly one token to each output channel per activation, and is skipped if any input is unavailable. We discuss two extensions: *multi-token reads and writes* for more compact models, and *optional inputs and outputs* for handling empty inputs.

Multi-Token Reads and Writes. Some applications require tasks to consume or produce multiple tokens per activation. For example, a down-sampling task may read several input samples to produce one output, while an up-sampling task may read one input to produce several outputs. We believe that these cases can be modeled using standard MIMOS, but to achieve more compact models, we allow each input channel to be annotated with a positive integer specifying the number of tokens read per activation, and similarly each output channel with a positive integer specifying the number of tokens written per activation. The standard model corresponds to the case where all counts are 1. Upon release, a task is enabled if each input channel contains at least as many tokens as its annotated count; otherwise, the release is skipped. It reads that many tokens from each input channel as a sequence and at its deadline instant appends as many tokens as its annotated count to each output channel. In MIMOS-Tools, the annotated count $k \in \mathbb{N}_{>0}$ is denoted by $[k]$.

Optional Inputs and Outputs. In some applications, a task may execute even when certain input channels are empty. For instance, a controller may proceed with default or previous values when sensor data is missing. To support this, we extend the domain of task functions to accept and produce absent values. Let $\mathsf{Val}_\perp = \mathsf{Val} \cup \{\perp\}$, where $\perp$ represents an absent value, similar to `Option` or `Maybe` types in functional programming [38]. A task function may then accept $\perp$ on some of its inputs, receiving it when the corresponding channel is empty with no token consumed. For those (and only those) input channels c whose corresponding argument in $F(\tau)$ admits $\perp$, τ is always enabled at release, much like register channels which always hold a valid value, even if c is empty; all other input channels must provide a value at release. Symmetrically, for those (and only those) output channels c whose corresponding result in $F(\tau)$ may be $\perp$, τ performs no write on c in an activation where $F(\tau)$ yields $\perp$ for c.

Naturally, these two extensions can be combined. For an input channel annotated with a *read-up-to* count $k \in \mathbb{N}_{>0}$, a read operation reads as many tokens as available, up to k, and the task function receives $\perp$ if the channel is empty. Such a channel does not affect whether a task is enabled at release. For an output channel annotated with a *write-up-to* count, a write operation may produce a sequence of up to k tokens. However, to preserve determinism, the exact number of tokens written must be determined by the computed output of its task function. In MIMOS-Tools, read-up-to and write-up-to counts are denoted by $[\leq k]$.

The Synchronous Case in MIMOS.. Recall that deadlines are assumed to be non-zero, a natural assumption for the schedulability of MIMOS models. However, to model synchronous systems under zero-time semantics or the synchronous hypothesis, we may set all task deadlines to zero and modify the transition rules to allow read, compute, and write operations to occur within the same time instant, following the causality relation defined by the MIMOS network. This yields a synchronous model consistent with those supported by existing tools, such as Simulink for synchronous design. This feature is currently supported by MIMOS-Tools, though technical details are deferred to future work.

3 Design Principles and Tool Architecture

This section briefly presents the design principles underlying the model of computation, which enable separation of concerns to increase modeling and implementation flexibility while reducing analysis complexity, and explains how these principles support the MIMOS-Tools toolchain architecture, from system modeling to code generation for deployment on heterogeneous platforms.

Functional Correctness Independent of Non-functional Behavior. Functional correctness should be preserved during design-space exploration aimed at satisfying non-functional requirements. For example, changing the execution time of a task must not affect its output or logical correctness. While this property is straightforward for an individual task, it becomes considerably more challenging at the system level, where functionality emerges from the coordinated execution of multiple tasks. The designer must ensure that variations in the execution order imposed by the scheduler do not alter the system's functional behavior. This ideal separation is achieved in the Kahn model, where functionality is uniquely defined and independent of the execution order of the Kahn processes. This is the essential reason why the MIMOS model is built on the Kahn model.

More precisely, MIMOS enforces this separation through a time-triggered execution pattern [27]. Each task reads its inputs at its release instant, computes its outputs before the deadline, and writes the outputs at the deadline, following a deterministic read–compute–write pattern. This abstracts away variations in a task's response time—an approach known as Logical Execution Time (LET), as adopted in Giotto [21]. Because all read and write operations occur at time instants determined solely by task periods and deadlines, data exchanged between tasks is independent of the physical scheduling of computations. This separation allows functional and timing properties to be verified independently, thereby reducing overall verification complexity.

Separation of Communication and Computation. Embedded systems face two fundamental concerns: computation and communication. Computational tasks should be designed independently of any specific communication mechanism. Allowing communication to occur during task execution introduces not only data races but also implicit interference: a task may receive new data mid-execution, altering its workload or control flow and tightly coupling producers and consumers. Such coupling complicates scheduling analysis. Unlike Kahn Process Networks [24], which rely on blocking reads, MIMOS tasks communicate exclusively through non-blocking, wait-free channels. Since tasks never stall waiting for input, system components can be specified as *independent real-time tasks*, enabling efficient and predictable timing analysis [14].

Layered Architecture. To decouple the logical system description from its physical deployment, MIMOS-Tools adopts a layered architecture inspired by standards such as AUTOSAR [3] and SAVOIR [23]. Unlike AUTOSAR, which primarily separates application software from the runtime environment and basic software services, MIMOS-Tools organizes its layers around the *engineering concerns* at each stage: functional correctness (specified by MIMOS models representing timed dataflow networks), timing correctness (ensured by schedulable multiple software threads), and platform deployment (executed dynamically on heterogeneous resources).

This layered approach, illustrated in Figure 2, is designed to enable flexible modeling, facilitate efficient analysis through abstraction, and provide re-configurability and adaptability through on-line flexible (not necessarily off-line static) scheduling [47].

Functional Layer: At the top is the *functional layer*, where systems are modeled as MIMOS networks. At this layer, we focus exclusively on functional correctness: designers specify task behaviors and channel connections without concern for schedulability or resource allocation. The toolchain at this layer supports graphical modeling and visualized simulation, static analysis for deadlock freedom checking and channel overflow detection, and functional verification. The deterministic semantics of MIMOS ensure that functional correctness established at this layer is preserved throughout the design flow.

Software Layer: The middle layer is the *software layer*, which puts the separation of communication and computation into practice. Leveraging the non-blocking communication, the verified components from the Functional Layer will be mapped into a set of independent real-time tasks. Each node can be abstracted as a DAG task [44] that exposes internal parallelism and dependencies among sub-functions. More generally, to optimize the final implementation, a group of nodes may be mapped to a single DAG task if their timing parameters fit together, e.g. with the same period. The toolchain at this layer supports scheduling and timing analysis to determine whether the task set is schedulable when given a scheduling policy and a platform configuration. This layer also addresses end-to-end latency analysis, i.e., the delay from an input event at one task to its observable effect at an output after propagation through intermediate tasks.

Hardware Layer: At the bottom is the *hardware layer*, where the validated system model is deployed onto target platforms. The code generation module automatically transforms the system model into executable programs. The generated code preserves the deterministic semantics established at the functional layer while respecting the timing constraints verified at the software layer.

This layered architecture enables designers to reason about functional correctness independently of timing concerns, then

progressively refine the design to meet real-time requirements without invalidating functional properties. Changes to timing parameters or scheduling policies at the software layer do not affect the functional behavior validated at the function layer, supporting efficient design-space exploration for safety-critical systems.

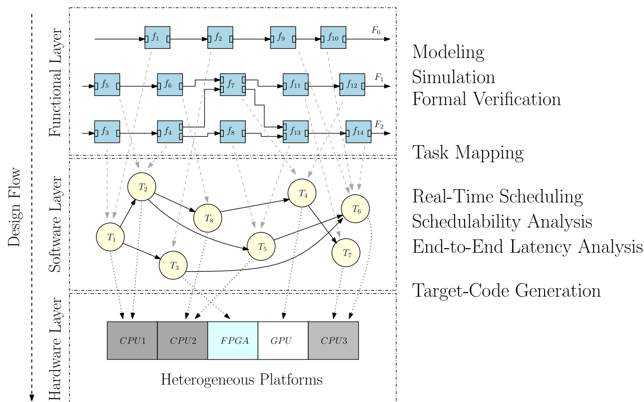


Fig. 2 The layered architecture of MIMOS-Tools and the key features of its individual tools. Left-to-right arrows indicate data flows through communication channels, while dashed top-to-bottom arrows represent task mapping (currently, to be provided by users) and real-time scheduling.

4 Key Features of MIMOS-Tools

MIMOS-Tools is composed of a set of individual tools, each offering dedicated functionality to support different stages of the system development process. In this section, we present its main features. To illustrate these features, we use a simplified Unmanned Aerial Vehicle (UAV) controller [32], a drone altitude controller as a running example throughout the section. The example system consists of four tasks. The `RadioControl` processes radio signals received from the pilot and issues target-altitude commands to the PID controller, thereby acting as a low-rate mission planner. The `PIDController` computes the control output based on the pilot's commands and the current altitude provided by the sensor. The `Sensor` samples the drone's current altitude, and the `Aerodynamics` models the physical dynamics of the UAV and acts as the actuator. Using this example, we demonstrate how MIMOS-Tools supports each stage of the development process.

4.1 Modeling and Simulation

The toolchain provides a graphical modeling environment that enables users to describe MIMOS models via a GUI (Figure 3). Users construct a system model by creating tasks and channels in a drag-and-drop manner. Each task can be configured through the properties panel with its timing parameters, associated channels, and channel types (FIFO or register). Tasks are interconnected by channels, thereby defining the communication structure of the system.

Hierarchical modeling. To support structural abstraction, the tool also provides two complementary mechanisms for hierarchical model composition and decomposition. First, a group of tasks and channels can be collapsed into a *subsystem*, a named visual block that hides the internal structure and helps manage visual complexity in large designs. Collapsing or expanding a subsystem does not alter the semantics of the model. Second, individual tasks can be defined as *composite nodes*, in which the task's functionality is internally organized as a directed acyclic graph (DAG) of sub-functions. This exposes internal parallelism that can be exploited by the scheduling and code generation stages on multi-core platforms. Unlike subsystems, a composite node represents a single MIMOS task and follows the same read–compute–write timing pattern described in subsection 2.1.

Mixed-language and LLM-assisted development. Once the system architecture has been defined, the user provides an implementation for the function of each task. The tool currently supports implementations in C++ and Java, and will be extended to additional programming languages to enable rapid prototyping of mixed-language systems. Beyond manual coding, the tool also supports *natural-language specification* of task functions. In this mode, a large language model (LLM) automatically generates implementations from user-provided descriptions. This capability is particularly effective for MIMOS tasks because each task is inherently a pure function over data streams.[8] Users therefore only need to describe *what* the function computes from its inputs, without having to manage system-wide dependencies.

Simulation and Visualization. The tool simulates models by executing the operational semantics defined in subsection 2.2 step by step. Users can specify the simulation duration and visualize the outputs of selected tasks as functions of time. The simulator progresses through successive event instants, producing a deterministic execution trajectory that is uniquely determined by the initial configuration and the system model.

User-defined *predicates* can be evaluated over the resulting trajectory. Each predicate is a Boolean function over configurations, and when applied to the execution sequence, it produces a corresponding sequence of truth values, enabling users to monitor whether system properties hold at each time instant. Figure 4 illustrates an example of LLM-assisted

[8] Local state can be viewed as a self-feedback loop: the updated state produced by one activation becomes an input to the next activation of the same task.

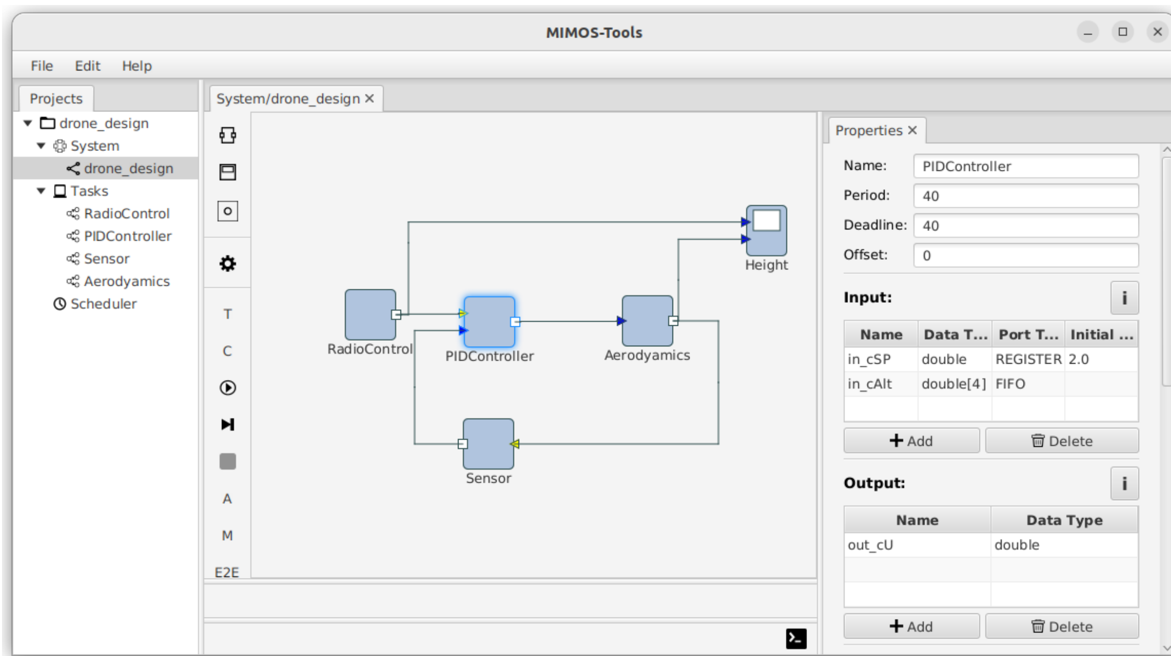


Fig. 3 Graphical modeling interface with properties panel of the UAV altitude controller, consisting of four tasks connected by FIFO and REGISTER channels.

development of the UAV system along with its simulation results.

4.2 Formal Verification

The simulation feature of MIMOS-Tools is intended to explore a finite portion of the reachable state space of a model over a bounded time horizon. It illustrates the operational behavior of the model by generating a finite execution trajectory. Logical properties can then be evaluated over such trajectories, for example, that the size of a FIFO queue remains below a fixed bound or that the value of a data variable stays within a given interval. However, simulation alone cannot establish whether these properties hold over the entire reachable state space of a MIMOS model (over potentially unbounded time). The verification feature, in contrast, aims to establish correctness properties for all execution trajectories, including infinite ones, by analyzing the static structure of the model and also by exhaustively exploring the reachable state space.

We distinguish two classes of correctness properties: (1) model consistency and feasibility, which can be checked using static techniques; and (2) local and global behavioral properties, which are verified through state-space exploration.

Deadlock and Buffer Overflow Detection. The tool provides a static analysis to verify *deadlock-freedom* and *overflow-freedom* of a model prior to deployment. These properties capture the *consistency* and *feasibility* of the model by reasoning about long-run token production and consumption rates using only structural information: the network topology, task periods and deadlines, read policies (e.g., the number of tokens required by a non-blocking read), and the initial token distribution.

A *deadlock* occurs when a task becomes permanently inactive because cyclic dependencies, combined with insufficient initial tokens, eventually starve one of its FIFO reads. A *channel overflow* occurs when the token count of a FIFO grows without bound because the producer's long-term execution rate exceeds that of the consumer. If neither problem is detected, the tool reports (i) the *task utilization*—the ratio of successful activations to total releases in the long run—and (ii) the *peak queue size* for each FIFO, which gives the minimal buffer capacity required for implementation. A utilization of 1.0 means that the task executes at every release, whereas lower values indicate inefficiencies such as suboptimal initial token placement or mismatched production and consumption rates. The computed buffer bounds are passed directly to the code generator (subsection 4.4) for static channel allocation. Figure 5 demonstrates these capabilities in modeling the UAV controller using MIMOS. The static analysis tool detected deadlock- and channel overflow-errors for two early versions of the model. By adjusting task parameters and channel types, a corrected model is achieved with the errors removed.

Specification and Verification of Functional Properties. Local correctness properties of a model can be specified as predicates over the inputs, local states, and outputs of its nodes. Global system-level properties are specified as

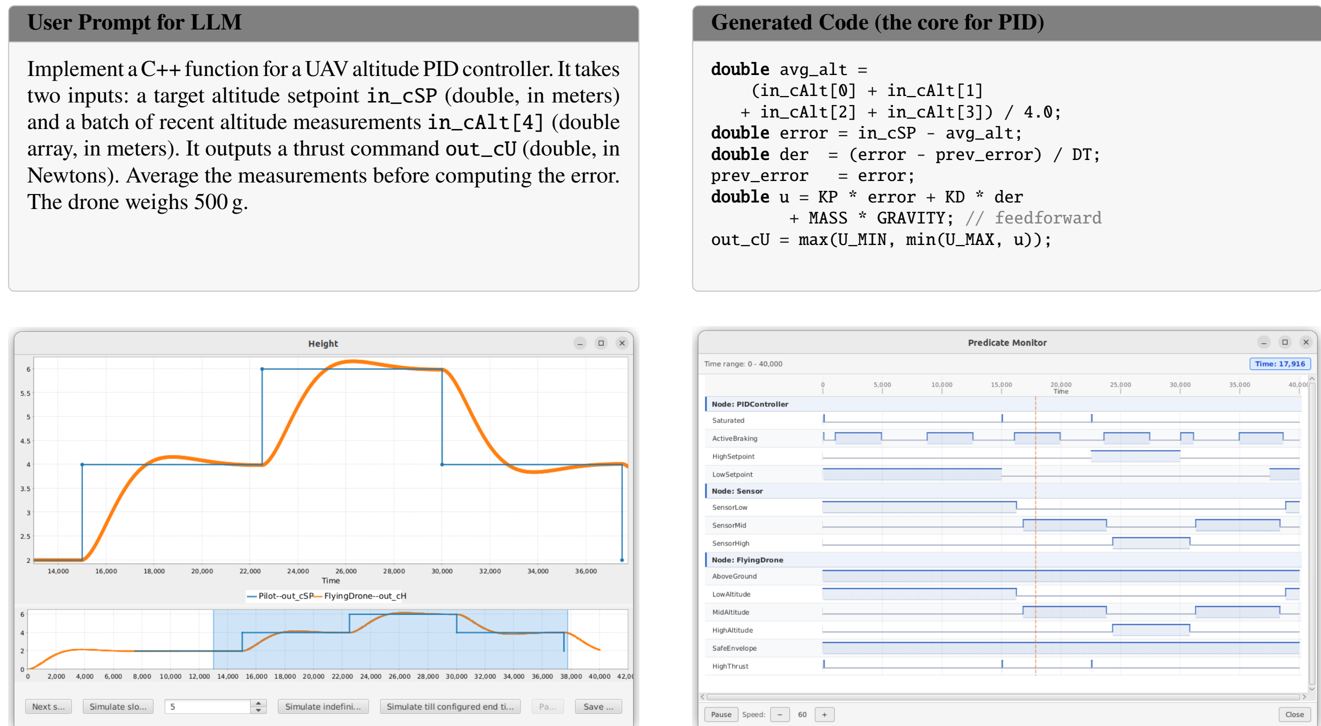
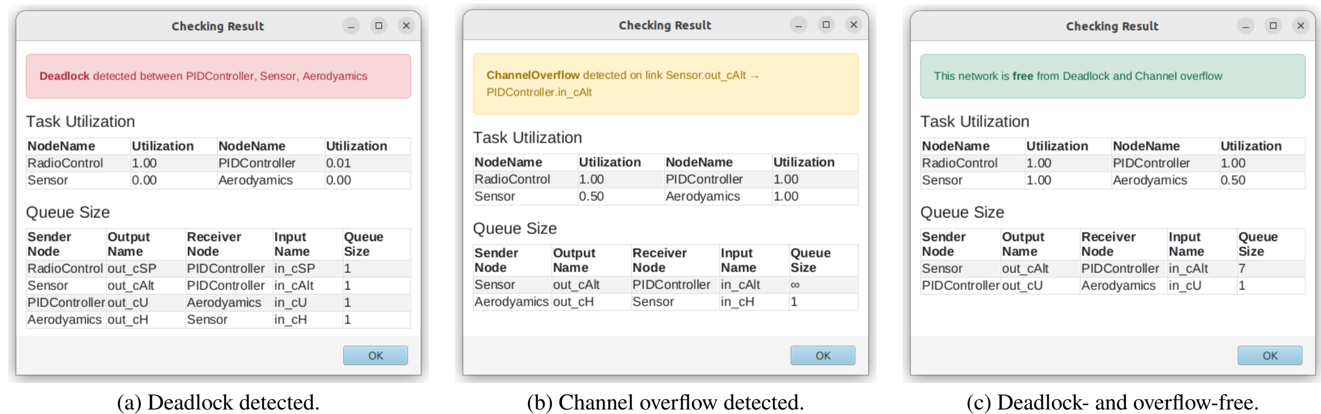


Fig. 4 LLM-assisted development and simulation of the UAV altitude controller. `Top left`: LLM-assisted task function generation — the user provides a natural language description of the task function for LLM. `Top right`: the tool generates the corresponding C++ implementation. `Bottom left`: simulated altitude tracking across four setpoints (2 m → 4 m → 6 m → 4 m). `Bottom right`: visualized true values of predicates of interests e.g. `SafeEnvelope` specifying a safety property, evaluated over the execution trajectory.



(a) Deadlock detected. (b) Channel overflow detected. (c) Deadlock- and overflow-free.

Fig. 5 Static analysis results for the UAV model: deadlock and channel overflow detected in early versions of the model by static analysis shown in (a) and (b), and resolved by adjusting timing parameters of tasks and channel types with correct results shown in (c).

temporal formulas over the local predicates. Currently, the tool supports the CTL fragment adopted in UPPAAL [29], including invariant and reachability properties, as well as lead-to properties expressing that the satisfaction of one local predicate (or a set of predicates) at some time instant implies the eventual satisfaction of another predicate at a later time.

As shown in Figure 4.1. the truth values of local predicates may vary along execution trajectories, i.e., a property may hold or not at a given time instant. Given a vector of such local predicates, the verification tool explores the full reachable state space using the operational semantics developed in subsection 2.2. In parallel, it constructs a finite automaton abstraction whose configurations correspond to the vectors of truth values of the local predicates. As an example, consider the predicates P:`Height > 40` and Q:`Height < 100`, which constrain the altitude of a flying drone. These predicates induce an automaton with four states over the full

state space of the drone model, corresponding to the possible combinations of their truth values.

Essentially the finite automaton abstraction is used to model check global temporal properties. Note that although the automaton is finite-state, the verification procedure must, in general, explore the full (potentially infinite unless the data domain of variables and also the length of FIFO's are bounded) state-space of the underlying model. The current implementation provides two verification techniques: explicit state-space exploration for rapid prototyping, and symbolic state-space representation and exploration based on abstract interpretation [12, 13].

Figure 6 shows the finite automaton constructed for the UAV controller, where 13 local predicates are defined. Although the number of abstract states grows exponentially with the number of predicates in the worst case, the automaton constructed from the UAV model remains compact and tractable, demonstrating the practical effectiveness of the abstraction.

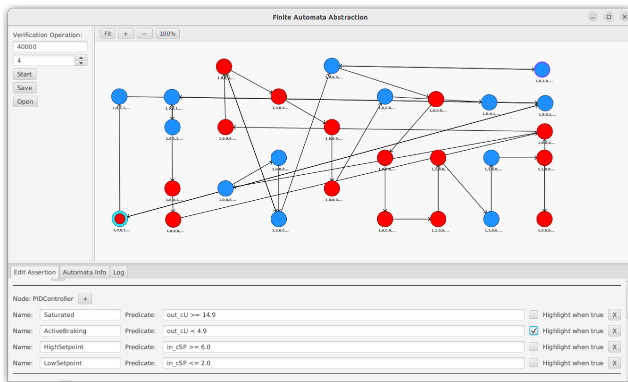


Fig. 6 Finite automata abstraction for 13 local predicates inserted in the UAV model, where the marked red-colored nodes highlight that particular predicates of interests, e.g, `out_cU < 4.9` (selected in the panel for specifying properties) are true of the model. The abstraction induces a substantially smaller automaton, illustrating the power of the verification technique of MIMOS-Tools.

4.3 Real-Time Scheduling and Analysis

This is one of the major features that distinguishes MIMOS-Tools from existing model-based design tools such as Simulink: MIMOS-Tools enables the specification of system components as real-time tasks, supports schedulability checking, and provides worst-case bounds on end-to-end latency from inputs to their corresponding outputs.

Task Abstraction of MIMOS Models. In the MIMOS model, the nodes (components) of a network represent real-time tasks. In MIMOS-Tools, each node is abstracted as a periodic real-time task characterized by its release period, relative deadline, and resource requirement, expressed as the worst-case execution time (WCET). In general, a node may be specified as a periodic DAG task, where vertices correspond to sub-functions with individual resource requirements. A sequential task is the special case where the DAG reduces to a single node. The WCET of each sub-function can be validated using a separate timing-analysis tool, such as AbsInt's aiT [15].

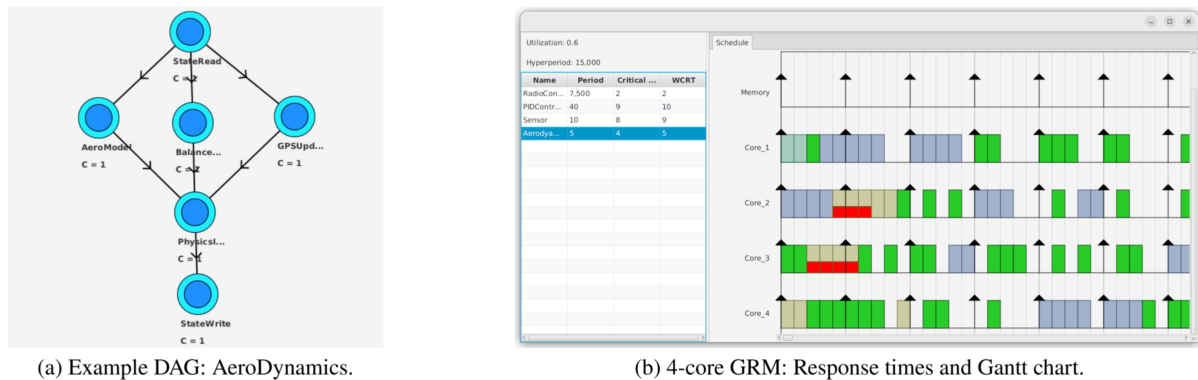
As described in section 3, system components in MIMOS communicate exclusively through non-blocking, wait-free channels. Because tasks never stall waiting for data, they can be treated as independent real-time tasks, for which efficient schedulability analysis techniques are available. The current implementation can handle systems with many thousands of nodes [36].

Scheduling and Schedulability Analysis. The designer configures the tool by specifying (1) the scheduling policy and preemption mode, (2) the platform parameters (number of cores, memory-access time, preemption cost, and communication cost), and (3) the analysis method. The tool currently supports more than 20 multi-core scheduling policies, grouped into three main categories: *global scheduling* (Global RM, Global FP, and Global EDF), *partitioned scheduling* with various bin-packing heuristics (worst-fit, best-fit, and first-fit), and *federated scheduling* variants.

The tool provides two analysis modes. The analytical mode computes schedulability guarantees and worst-case response times (WCRT) using established methods tailored to the selected scheduling policy. The simulation mode visually simulates task execution over the system's hyperperiod, checks for deadline violations, and extracts the observed worst-case response times.

We illustrate the scheduling feature with the UAV controller. Figure 7(a) shows an example DAG for the `AeroDynamics` task of the controller. Figure 7(b) shows the schedule on a 4-core platform under Global RM policy and also the worst-case response times of the four main tasks of the controller. The left panel reports each task's timing parameters, and computed WCRT (Worst-Case Response Times).

End-to-End Latency Analysis. Beyond response times, the tool also computes *worst-case end-to-end latency* for user-specified source–sink paths through the MIMOS network. Intuitively, the end-to-end latency of a path measures the maximum delay from a data change at a source task's input to its observable effect at a sink task's output, after propagation through all intermediate tasks and channels [48]. Because task reads and writes occur at deterministic time instants fixed by periods and deadlines, these latencies depend solely on the network topology and task timing parameters, and are independent of the scheduling policy and platform configuration. The tool reports the worst-case delay for each specified path.



(a) Example DAG: AeroDynamics.

(b) 4-core GRM: Response times and Gantt chart.

Fig. 7 Real-time scheduling and schedulability analysis for the UAV model with 4 DAG tasks scheduled by Global RM (GRM) scheduling.

A concrete demonstration on a multi-rate avionics system is given in subsection 5.1.

4.4 Code Generation

Consistent (deadlock-free), feasible (overflow-free), timing-correct (with schedulable tasks and task chains meeting end-to-end timing constraints), and functionally correct design models can be obtained after the previous design steps. These models can then be transformed into executable programs while preserving their specified execution semantics and verified timing and functional properties, targeting multi-core platforms running *Real-time Linux* [39].

The resulting executable is organized into three layers: (i) *Node functionality*, comprising the task functions defined by the user during modeling (subsection 4.1); (ii) *Coordination*, managing inter-task communication through channels and enforcing correct read/write semantics as defined in subsection 2.2; and (iii) *Scheduling policy*, orchestrating task execution across processor cores according to the analyzed scheduling parameters (subsection 4.3). Note that timing properties, such as task schedulability, are verified with respect to this scheduling policy. The coordination layer instantiates channels with statically determined buffer sizes obtained from the verification phase (subsection 4.2), ensuring that the generated buffers are both sufficient and memory-efficient. The code generator automatically produces layers (ii) and (iii) and integrates them with the user-defined task functions. Figure 8 shows representative snapshots of the coordination and scheduling layers for the UAV example.

5 Case Studies

In the previous section, we used the UAV controller to demonstrate how all features of MIMOS-Tools, covering the entire development process from modeling to automatic code generation for deployment. This section presents case studies selected from different application domains and illustrate how they can be modeled and validated with a focus on specific aspects: modeling and simulation, formal verification, and real-time scheduling.

5.1 ROSACE Longitudinal Flight Controller

ROSACE (Research Open-Source Avionics and Control Engineering) [37] is a well-established multi-periodic longitudinal flight controller which has been studied extensively in the literature [7, 41, 31]. As described in [37], it regulates altitude h, vertical speed V_z, and airspeed V_a for a medium-range civil aircraft. Compared to the simplified UAV controller introduced in section 4, this case study shows how the toolchain handles a complex multi-rate system from industrial avionics.

The MIMOS model (Figure 9(a)) comprises 13 tasks distributed across four rates: three environment-simulation tasks (`Elevator`, `Engine`, `AircraftDynamics`) at 200 Hz, five sensor-filter tasks at 100 Hz, three control-law tasks at 50 Hz, and two command-input tasks (`speedCommand` at 5 Hz and `heightCommand` at 1 Hz). `Elevator` and `Engine` start at offset = 0 ms with a deadline of 1 ms, followed by `AircraftDynamics` (offset = 1 ms, deadline = 1 ms), the five sensor filters (offset = 2 ms; `hFilter` with deadline = 1 ms, `azFilter`, `VzFilter`, `VaFilter`, and `qFilter` with deadline = 2 ms), `AltitudeHold` (offset = 3 ms, deadline = 1 ms), and `VerticalSpeedControl` and `TrueAirspeedControl` (offset = 4 ms, deadline = 1 ms). Register channels (yellow arrows) capture multi-rate sensor sampling, FIFO channels (blue arrows) enforce data-dependency ordering for command flows, and FIFO channels with optional consumer input (red arrows) prevent fast actuators from blocking on slower controllers.

Figure 9(b)–(d) show the results for a sequential altitude-change scenario (10,000 m → 10,500 m → 11,000 m → 10,500 m) at constant airspeed. The controller exhibits the expected behavior: altitude tracks each setpoint with minimal overshoot, vertical speed is regulated at the commanded ±2.5 m/s, and airspeed remains close

Listing 1 Generated Coordination Layer

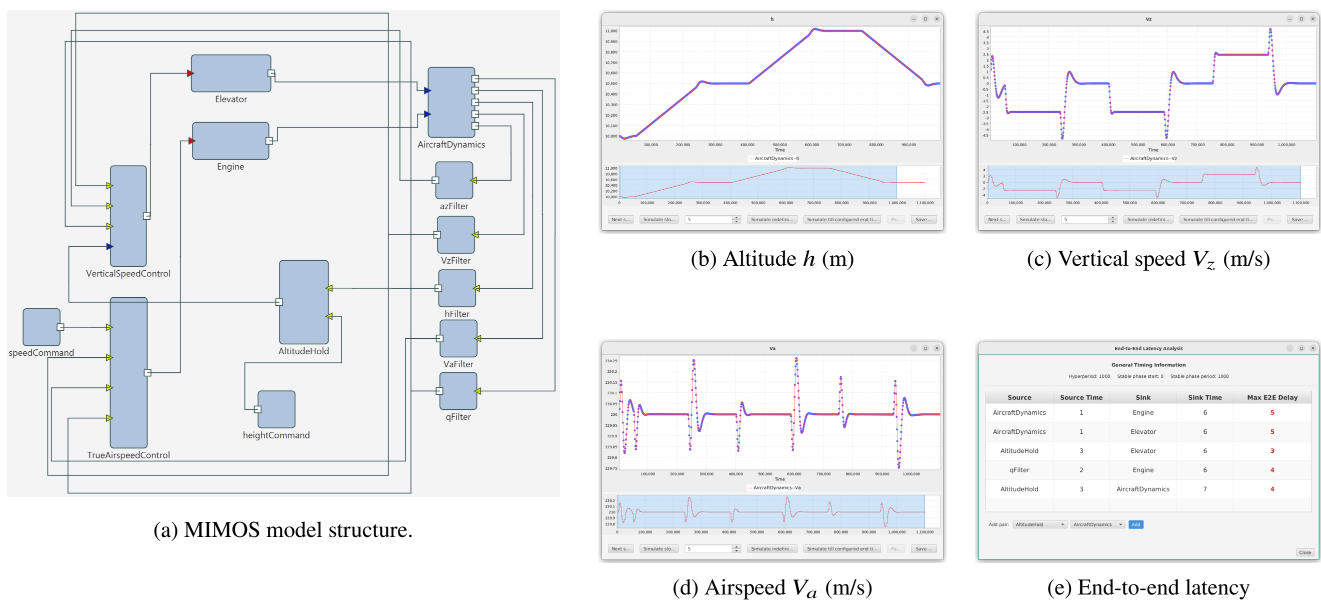
```
//   Read inputs at the release instant
void readTaskPIDController(long long ts) {
  pIDController.set_in_cSP(
      *PilotPIDControllerin_cSP.pop(ts));
  SensorPIDControllerin_cAlt.tryPop(
      [&](double *msg) {
          pIDController.set_in_cAlt(msg); });
}
//   Write outputs at the deadline instant
void writeTaskPIDController(long long ts) {
  sendData(&PIDControllerFlyingDronein_cU,
           pIDController.get_out_cU(), ts);
}
// Enablement check: INACTIVE if no data expected,
// WAITING if upstream write still in progress
TaskState taskPIDControllerOrWait(
    long long ts, long long wts) {
  auto state = getPIDControllerState(ts);
  if (state & INACTIVE) return INACTIVE;
  if (state & WAITING)  return WAITING;
  setWritingStateTaskPIDController(wts);
  readTaskPIDController(ts);  // read at release
  pIDController.run();        // user computation
  return ACTIVATED;
}
```

Listing 2 Generated Scheduling Policy

```
// Task           Period    Priority
// Sensor          10 ms      80
// FlyingDrone     20 ms      79
// PIDController   40 ms      78
// Pilot         3500 ms      77

PeriodicTask ptPIDController(
  PIDCONTROLLER_PERIOD,PIDCONTROLLER_OFFSET,
  PIDCONTROLLER_CPU_ID,PIDCONTROLLER_PRIORITY,
  PIDCONTROLLER_POLICY,PIDCONTROLLER_IS_REAL_TIME,
  PIDCONTROLLER_RUN_TIME);
ptPIDController.start(
  &startApplicationTime, startUpTime,
  // Check at release; WAITING if upstream
  // write is still in progress
  [this](long long t, long long w) {
    return taskPIDControllerOrWait(t, w); },
  // Bounded re-check, skip if still absent
  [this](long long t, long long w) {
    return taskPIDController(t, w); },
  // Write outputs at the deadline instant
  [this](long long t) {
    writeTaskPIDController(t); });
```

Fig. 8 Selected parts of the generated code in C++ for the UAV model. `Left`: the coordination layer generated enforces inputs are read at the release instants and outputs written at the deadline. It checks the availability of inputs: it returns `INACTIVE` if no data is expected in this period, or `WAITING` if an upstream task has not yet completed its write. `Right`: the generated scheduling policy uses the Rate-Monotonic priorities from the scheduling analysis and passes them into the PeriodicTask constructor. Each task is then launched with three callbacks produced by the coordination layer.



Source	Source Time	Sink	Sink Time	Max E2E Delay
AircraftDynamics	1	Engine	6	5
AircraftDynamics	1	Elevator	6	5
AltitudeHold	3	Elevator	6	3
qFilter	2	Engine	6	4
AltitudeHold	3	AircraftDynamics	7	4

(a) MIMOS model structure.

(b) Altitude h (m)

(c) Vertical speed V_z (m/s)

(d) Airspeed V_a (m/s)

(e) End-to-end latency

Fig. 9 ROSACE Case Study with MIMOS-Tools: (a) communication structure; (b)–(d) simulation results for altitude, vertical speed, and airspeed; (e) end-to-end latency analysis.

to the nominal 230 m/s throughout the maneuver. Figure 9(e) shows the end-to-end latency for selected source–sink pairs. The critical control path `AltitudeHold` → `Elevator` exhibits a worst-case delay of 3 ms, while the full plant-feedback paths `AircraftDynamics` → `Engine` and `AircraftDynamics` → `Elevator` both reach 5 ms—meeting the original ROSACE requirement that the entire control chain completes within 5 ms every 20 ms [37].

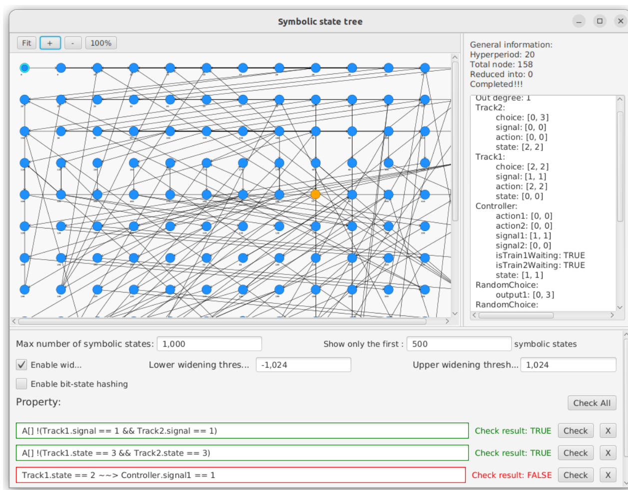


Fig. 10 Verification of the train crossing controller [1]. MIMOS-Tools produces a finite automaton with 158 states, representing the system that is potentially infinite-state.

5.2 Train Crossing Control

The train crossing controller is a classic benchmark in the formal verification community, originally proposed for the UPPAAL model checker [49, 29], and adopted later in different contexts e.g. embedded systems programming [1]. The case study here is based on the version in [1], where the controller manages two (potentially infinite) streams of trains from two separate railway tracks, merging them onto one shared track for crossing e.g. a bridge.

A train (`Track1` or `Track2`) operates as a state machine with four states: (0) far away, (1) approaching, (2) stopped, and (3) crossing. The controller maintains three states: (0) free, (1) occupied by Train 1, and (2) occupied by Train 2. The controller's output `signal` to each train is either red (stop and wait) or green (proceed to cross). When a train approaches, it sends an `action` signal to the controller; the controller grants access by setting the corresponding `signal` to green if the shared track is free, otherwise the train waits in the stopped state. The controller implements a fixed priority policy: when both trains are waiting, Train 1 is always granted access first. The critical safety requirement is collision avoidance: two trains must never occupy the shared crossing track simultaneously.

Note that the system is essentially infinite state, as the two streams of trains are driven by independent random choices, arriving and leaving at different time points according to different patterns. Despite this, MIMOS-Tools reduces the system to a finite automaton of 158 distinct symbolic states, each representing an equivalence class of concrete states, enabling exhaustive verification.

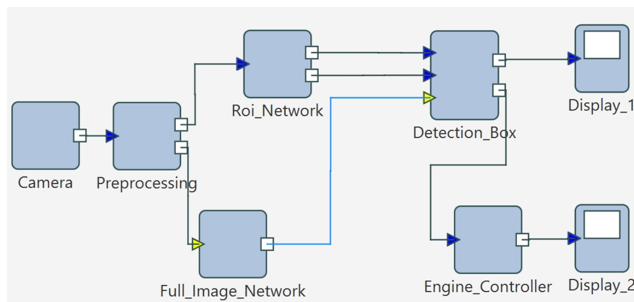
Three safety properties are verified against this model. The first two hold: *mutual exclusion* `A[] !(Track1.signal == 1 && Track2.signal == 1)` ensures the controller never grants green signals to both trains simultaneously, and *collision freedom* `A[] !(Track1.state == 3 && Track2.state == 3)` guarantees both trains never occupy the crossing at the same time. However, the third property—*responsiveness* `Track1.state == 2 -> Controller.signal1 == 1`—is verified FALSE, revealing a liveness issue: a train stopped at the crossing entrance may never receive a green signal, leading to indefinite waiting. This subtle design flaw is automatically detected through exhaustive symbolic exploration.

5.3 Real-Time Perception-Action Pipeline for Autonomous Vehicles

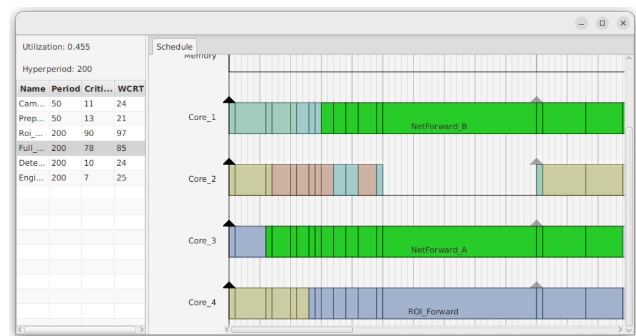
This case study is from automotive industry, studied in [25]. It is a subsystem for Autonomous urban driving with hard real-time constraints to detect and classify objects in traffics, and accordingly, make driving decisions. Deep Neural Network (DNN) inference has become the standard approach for object detection, but it is computationally expensive, making hard real-time deployment challenging. One way to reduce inference cost without compromising safety is to prioritize computation over the most critical parts of the image. It motivates a dual-path structure: a high-priority ROI (region of interest) path directly feeding the control decision, and a lower-priority full-scene path supplying contextual consistency checks.

We use it to demonstrate the modeling, scheduling, simulation, and monitoring capabilities of MIMOS-Tools. The system comprises six tasks shown in Figure 11(a): a `Camera` task acquires frames at 20 fps (period 50 ms), a `Preprocessing` task crops the ROI and produces a downscaled full-scene image at the same rate, a `Roi_Network` task runs every 100 ms with an internal fork-join structure over the two ROI halves, a `Full_Image_Network` task runs every 200 ms consuming the most recently available preprocessed frame via a Register channel, a `Detection_Box` task fuses ROI and scene-level detections every 100 ms, and an `Engine_Controller` task translates fused detections into speed commands for the drive-by-wire controller. The controller applies a three-level safety response policy: upon detecting a vehicle or a cyclist, it decelerates to a reduced speed; upon detecting a pedestrian, it issues an emergency stop command, bringing the vehicle to a full halt until the pedestrian clears the road-ahead zone; otherwise, it maintains normal cruise speed.

Figure 11(b) shows the scheduling analysis result on a four-core platform under Global EDF. Figure 11(c) shows the simulation result at a representative time point: the pipeline successfully detects pedestrians, vehicles, and cyclists in the scene and overlays bounding boxes on the camera feed. When a pedestrian is detected in the road-ahead zone, `Engine_Controller` immediately issues an emergency stop

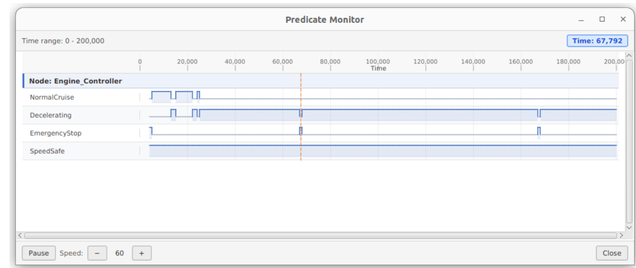


(a) The MIMOS Model for the urban autonomous system.



(b) Response times analysis and Gantt Chart.

(c) Simulation of traffic scenarios and Object-detection.



(d) Safety monitoring and control logic.

Fig. 11 The urban autonomous perception pipeline for self-driving vehicles: (a) MIMOS model; (b) scheduling analysis on a 4-core platform; (c) traffic simulation and detection results; (d) safety monitoring and emergency control triggers.

command, bringing the vehicle to a full halt—demonstrating that the safety-critical response is correctly triggered end-to-end through the pipeline. Figure 11(d) shows the predicate monitor attached to `Engine_Controller`, confirming that `SpeedSafe` holds almost continuously, `Decelerating` activates transiently during vehicle and cyclist encounters, and `EmergencyStop` fires only at isolated time points corresponding to pedestrian detection events—consistent with the defined safety policy.

6 Related Work and Comparison

The synchronous paradigm is the dominant solution widely used in safety-critical domains and underlies industrial tools such as Simulink and SCADE [43, 2], programming languages including Lustre, Esterel, Signal and Prelude [4, 20, 6, 5, 16], computational models e.g. Synchronous Data Flow (SDF) and extended variants [30, 18, 11]. The synchronous approach assumes zero-time computation and instantaneous communication in logical time, it provides deterministic functional semantics. In practice, systems in these frameworks are scheduled *statically* and implemented by mapping all executions to a global base rate, typically the greatest common divisor (GCD) of all task periods. The static schedule, communication pattern, and buffering strategy are then computed over the corresponding hyper-period. While effective on uniprocessors [10], this approach introduces tight global coupling: even a small local change, such as adding a task with a new period, requires recomputing the base rate, reconstructing the entire schedule and essentially redesigning the system. This limitation becomes more serious on modern embedded platforms, which are increasingly multicore, heterogeneous, and distributed [40]. Communication delays, shared-resource contention, and independent clocks already challenge the synchronous abstraction, and global base-rate implementations further reduce flexibility. For example, in the ROSACE case study [37], tasks with different periods are executed inside a main loop synchronized to a 5 ms base rate. Adding a task with period 11 ms forces the system tick to drop to 1 ms and increases the hyper-period from 100 ms to 1100 ms, requiring complete recomputation and reimplementation of the schedule and communication structure. Such rigid global coupling conflicts with the incremental evolution typical of large safety-critical systems [22, 46].

MIMOS addresses this problem by eliminating the global base rate. Tasks execute with their own periods and can be scheduled dynamically, while schedulability is analyzed offline. As a result, adding a new task does not affect existing ones as long as the system remains schedulable, providing true composability and enabling modular and scalable evolution.

Rooted in the time-triggered paradigm [27], the LET approach adopted by Giotto [21] fixes logical read and write

instants independently of the actual execution. MIMOS similarly employs LET to guarantee timing determinism by restricting channel reads and writes to predefined instants, such as deadlines. In addition, MIMOS uses FIFO queues to ensure functional determinism—a feature not provided by Giotto. Beyond ensuring functional determinism as in Kahn Process Networks [24], we believe that FIFO queues also help tolerate timing errors such as release jitter, a property we plan to investigate in future work.

Several alternative paradigms also aim to achieve determinism. The Bittide mechanism [28] and its associated programming model Timetide [26] enable deterministic distributed execution through logical synchrony. Another line of work focuses on deterministic computing, including PTIDES [50] and its successor Lingua Franca [34,35], which enforce functional determinism through timestamps and event ordering. However, in these approaches, logical time can diverge from wall-clock time, limiting their applicability for hard real-time systems with strict deadline guarantees. MIMOS-Tools targets real-time systems for which, achieving timing determinism remains more challenging and typically requires explicit scheduling and timing analysis.

7 Conclusion and Future Work

In this paper, we presented a new model of computation for the asynchronous design and implementation of embedded real-time systems, along with a toolchain based on this model that supports the full development lifecycle of such systems. All phases of the development process—including graphical modeling, simulation, verification, real-time scheduling, schedulability and end-to-end latency analysis, and automatic code generation for deployable executables on heterogeneous multicore platforms—are grounded in this single, coherent, and deterministic model of computation. This ensures that verified functional and timing properties are preserved throughout the development process and in the final implementation. Moreover, the toolchain supports modular and incremental design, implementation, and future updates, enabled by the underlying model for asynchronous communication and computation via asynchronous channels and flexible online scheduling.

The current version of MIMOS-Tools is ready for release and can be used for educational, research, and industrial purposes. However, developing a new toolchain requires substantial resources and ongoing efforts for testing, bug reporting and fixing, feature extension, and improvements in scalability and applicability, particularly for large-scale industrial systems. For future work, we plan to extend the LLM-assisted design capabilities beyond the generation of individual task functions. Currently, the toolchain uses LLMs to generate task implementations from natural language descriptions, exploiting the functional purity of MIMOS tasks. Future enhancements will focus on system-level design, enabling the automatic generation of complete MIMOS network architectures from high-level system requirements. We are mindful of the critical need for powerful verification techniques to provide safety guarantees in AI-assisted system development. We plan to improve the scalability of the symbolic verifier for state-space exploration. In particular, we aim to support the verification of Signal Temporal Logic (STL), which is currently employed only for monitoring. Furthermore, we intend to expand the range of platforms supported as targets for code generation.

References

1. Alur, R.: Principles of cyber-physical systems. MIT Press (2015)
2. Ansys, Inc.: SCADE Suite: model-based development environment for critical embedded software. https://www.ansys.com/products/embedded-software/ansys-scade-suite. Accessed: 2026-02-12
3. AUTOSAR: Specification of standard types. Tech. rep., AUTOSAR (2024). Available at https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_StandardTypes.pdf
4. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. Proceedings of the IEEE **79**(9), 1270–1282 (1991)
5. Benveniste, A., Guernic, P.L., Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming **16**(2), 103–149 (1991)
6. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming **19**(2), 87–152 (1992)
7. Bourke, T., Bregeon, V., Pouzet, M.: Scheduling and compiling rate-synchronous programs with end-to-end latency constraints. In: Euromicro Conference on Real-Time Systems (ECRTS) (2023)
8. Burns, A., Davis, R.I.: A survey of research into mixed-criticality systems. ACM Computing Surveys **50**(6), 82:1–82:37 (2017)
9. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications. Springer (2011)
10. Caspi, P., Scaife, N., Sofronis, C., Tripakis, S.: Semantics-preserving multitask implementation of synchronous programs. ACM Transactions on Embedded Computing Systems **7**(2), 15:1–15:40 (2008)
11. Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. Ph.D. thesis, Stanford University (1984)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) (1977)
13. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) (1979)
14. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys **43**(4), 1–44 (2011)
15. Ferdinand, C., Heckmann, R.: aiT: worst-case execution time prediction by static program analysis. In: Building the Information Society (IFIP), pp. 377–383 (2004)

16. Forget, J., Boniol, F., Lesens, D., Pagetti, C.: A multi-periodic synchronous data-flow language. In: IEEE High Assurance Systems Engineering Symposium (HASE), pp. 251–260 (2008)
17. Fritzson, P., Engelson, V.: Modelica: a unified object-oriented language for system modeling and simulation. In: European Conference on Object-Oriented Programming (ECOOP) (1998)
18. Ghamarian, A.H., Geilen, M.C.W., Stuijk, S., Basten, T., Theelen, B.D., Mousavi, M.R., Moonen, A.J.M., Bekooij, M.J.G.: Throughput analysis of synchronous data flow graphs. In: International Conference on Application of Concurrency to System Design (ACSD), pp. 25–36 (2006)
19. Guernic, P.L., Gautier, T., Borgne, M.L., Maire, C.L.: Programming real-time applications with SIGNAL. Proceedings of the IEEE **79**(9), 1321–1336 (1991)
20. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
21. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE **91**(1), 84–99 (2003)
22. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: International Symposium on Formal Methods (FM) (2006)
23. Hult, T.: SAVOIR handbook. ADCSS presentation, European Space Agency (ESA) (2016). Available at https://indico.esa.int/event/148/contributions/924/attachments/1030/1221/04-SAVOIR_handbook_ADCSS2016.pdf
24. Kahn, G.: The semantics of a simple language for parallel programming. In: IFIP Congress (1974)
25. Kang, W., Chung, S., Kim, J.Y., Lee, Y., Lee, K., Lee, J., Shin, K.G., Chwa, H.S.: DNN-SAM: split-and-merge DNN execution for real-time object detection. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2022)
26. Kenwright, L., Roop, P.S., Allen, N., Cačsaval, C., Malik, A.: Timetide: a programming model for logically synchronous distributed systems. ACM Transactions on Embedded Computing Systems **24**(5s), 100:1–100:25 (2025)
27. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE **91**(1), 112–126 (2003)
28. Lall, S., Cačsaval, C., Izzard, M., Spalink, T.: Logical synchrony and the Bittide mechanism. IEEE Transactions on Parallel and Distributed Systems **35**(11), 1936–1948 (2024)
29. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997)
30. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
31. Lee, E.A., Saussié, D., Pagetti, C.: Aircraft controller: the ROSACE case study. Lingua Franca Playground (2023). Available at https://github.com/lf-lang/playground-lingua-franca/tree/main/examples/C/src/rosace
32. Levy, S.D.: Robustness through simplicity: A minimalist gateway to neurorobotic flight. Frontiers in Neurorobotics **14**, 16 (2020).
33. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1), 46–61 (1973)
34. Lohstroh, M., Bateni, S., Menard, C., Schulz-Rosengarten, A., Castrillon, J., Lee, E.A.: Deterministic coordination across multiple timelines. ACM Transactions on Embedded Computing Systems **23**(5), 77:1–77:29 (2024)
35. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems **20**(4), 36:1–36:27 (2021)
36. Mohaqeqi, M., Dai, G., Khodabandeloo, B., Voudouris, P., Yi, W.: A scheduling and analysis tool for parallel real-time applications on multicore platforms. In: RTSS@Work: Open Demo Session of the IEEE Real-Time Systems Symposium (RTSS) (2022). Demo paper
37. Pagetti, C., Saussié, D., Gratia, R., Noulard, E., Siron, P.: The ROSACE case study: from Simulink specification to multi/many-core execution. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2014)
38. Pierce, B.C.: Types and programming languages. MIT Press (2002)
39. Reghenzani, F., Massari, G., Fornaciari, W.: The real-time Linux kernel: a survey on PREEMPT_RT. ACM Computing Surveys **52**(1), 1–36 (2019)
40. Saidi, S., Ernst, R., Uhrig, S., Theiling, H., de Dinechin, B.D.: The shift to multicores in real-time and safety-critical systems. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2015)
41. Schuh, M., Maiza, C., Goossens, J., Raymond, P., de Dinechin, B.D.: A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory. In: IEEE Real-Time Systems Symposium (RTSS) (2020)
42. Stigge, M., Yi, W.: Graph-based models for real-time workload: a survey. Real-Time Systems **51**(5), 602–636 (2015)
43. The MathWorks, Inc.: Simulink: simulation and model-based design. https://www.mathworks.com/products/simulink.html. Accessed: 2026-02-12
44. Wang, Y., Guan, N., Sun, J., Lv, M., He, Q., He, T., Yi, W.: Benchmarking OpenMP programs for real-time scheduling. In: IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2017)
45. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems **7**(3), 1–53 (2008)
46. Yi, W.: Towards customizable cyber-physical systems: composability, efficiency and predictability. In: International Conference on Formal Engineering Methods (ICFEM) (2017)
47. Yi, W.: Design and dynamic update of real-time systems. In: IEEE Real-Time Systems Symposium (RTSS) (2019)
48. Yi, W., Mohaqeqi, M., Graf, S.: MIMOS: A deterministic model for the design and update of real-time systems. In: International Conference on Coordination Models and Languages (Coordination) (2022)
49. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: International Conference on Formal Description Techniques (FORTE), pp. 243–258 (1995)
50. Zhao, Y., Liu, J., Lee, E.A.: A programming model for time-synchronized distributed real-time systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2007)