International Journal on Software Tools for Technology Transfer (2024) 26:229–245
https://doi.org/10.1007/s10009-024-00743-4

GENERAL

Special Issue: FMICS 2022

Reusable formal models for concurrency and communication in custom real-time operating systems

Julius Adelt[1] · Julian Gebker[1] · Paula Herber[1]

Accepted: 6 February 2024 / Published online: 20 February 2024
© The Author(s) 2024

Abstract

In embedded systems, the execution semantics of the real-time operating system (RTOS), which is responsible for scheduling and timely execution of concurrent processes, is crucial for the correctness of the overall system. However, existing approaches for the formal verification of embedded systems typically abstract from the RTOS completely, or provide a detailed and synthesizable formal model of the RTOS. While the former may lead to unsafe systems, the latter is not compatible with industrial design processes. In this paper, we present an approach for reusable abstract formal models that can be configured for custom RTOS. Our key idea is to formally capture common execution mechanisms of RTOS like preemptive scheduling, event synchronization, and communication abstractly in configurable timed automata models. These abstract formal models can be configured for a concrete custom RTOS, and they can be combined into a formal system model together with a concrete application. Our reusable models significantly reduce the manual effort of defining a formal model that captures concurrency and real-time behavior, together with the functionality of an application. The resulting formal model enables analysis, verification, and graphical simulation. We validate our approach by formalizing and analyzing a rescue robot application running the custom open source RTOS EV3RT.

Keywords Real-time systems · Formal verification · Reusability

1 Introduction

In the embedded systems industry, many companies use their own custom real-time operating system (RTOS). The RTOS schedules concurrent processes, takes care of process interactions and shared resources, and is thus crucial for the synchronization and timing behavior. To ensure the correctness of embedded systems, it is vital to correctly capture and analyze concurrency and time. Existing approaches for the formal verification of embedded systems, however, either abstract from the underlying RTOS completely (e.g., CPAchecker [10], Frama-C [12]) or they provide a fully formalized and verified RTOS (e.g., Sel4 [21], CertiKOS [15]). While the former abstracts from the influence of the RTOS on concurrent, timing-dependent applications completely, the latter requires extremely high manual effort and expertise, as a new formalization is needed for each custom RTOS.

✉ J. Adelt
julius.adelt@uni-muenster.de

P. Herber
paula.herber@uni-muenster.de

[1] University of Münster, Münster, Germany

In this paper, we propose reusable abstract formal models that can be configured for custom RTOS. It is an extended version of [3]. The key idea of [3] is to formally capture common execution mechanisms of RTOS abstractly in configurable timed automata models. To achieve this, we abstractly formalize preemptive and non-preemptive execution, priority-based scheduling, general task management, event synchronizations, and sensor APIs. For a given custom RTOS, the designer can use our configurable timed automata models to build a formal model that defines the execution semantics of key RTOS components like the scheduler and tasks. Furthermore, if the designer defines a mapping from system calls to abstract execution mechanisms (e.g., task activations or event notifications), a given real-time application can be combined into a formal model together with the RTOS components. The resulting model captures the concurrent and real-time dependent behavior as well as the functionality of the application precisely, but abstracts from the implementation details of the custom RTOS. It can be analyzed and verified using existing tools for graphical simulation, formal verification, and timing analysis like, for example, the Uppaal tool suite.

To validate the applicability of our approach, we have formalized and analyzed a search and rescue robot application

running the custom open source RTOS EV3RT. To formalize this custom RTOS, we have configured our reusable abstract timed automata models with an appropriate scheduling strategy and mapped system calls to abstract execution mechanisms. We have manually translated the task implementations for our case study. The automation remains subject for future work.

Compared to [3], we make the following contributions in this paper:

– We provide configurable timed automata models for synchronous and asynchronous data queues. With that, we extend our approach for modeling real-time dependent concurrency in custom RTOS (using events and time) with communication through data queues (using the new abstract models).
– We illustrate the use of the communication models with our case study of a search and rescue robot.
– We give deeper insights into the application of our reusable formal models for preemptive scheduling, events and communication using a selected task from our case study.
– We have verified additional properties that validate our communication models.

As in [3], we have again exploited UPPAAL's simulation and graphical animation of counterexamples to validate the system's functionality and the task interactions on the resulting formal model without executing it on the real robot hardware. For a model with a fixed mission sequence, we have analyzed and verified crucial safety and timing properties using the UPPAAL model checker.

The rest of this paper is structured as follows: In Sect. 2, we introduce core components of real-time operating systems, including synchronous and asynchronous data queues, and UPPAAL timed automata. In Sect. 3, we introduce our approach to provide reusable formal models for concurrency and communication in custom RTOS. We present our case study in Sect. 4 and experimental results in Sect. 5. In Sect. 6, we discuss related work, and we conclude in Sect. 7.

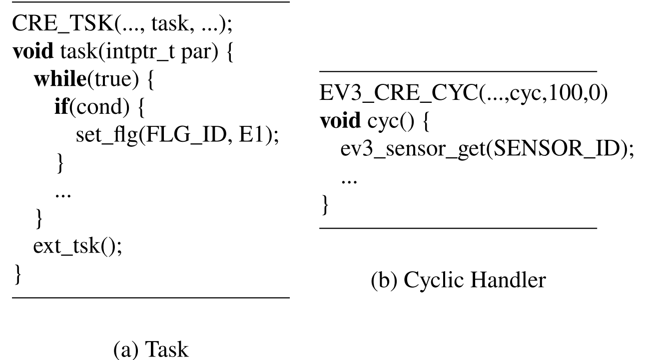
2 Preliminaries

In this section, we introduce preliminaries for the remainder of this paper, namely core components of real-time operating systems and UPPAAL timed automata.

2.1 Core components of real-time operating systems

There exists a large variety of custom RTOS. However, many of them follow certain standards, like OSEK/VDX [23] (in the automotive domain) or TOPPERS [26]. These standards informally define kernel objects, e.g., tasks, events, and resources, and their interactions. In the following, we briefly introduce kernel objects that can commonly be found in any RTOS implementation.

Tasks Tasks are typically the main execution unit in RTOS. While the instructions within each task are sequentially executed, tasks are concurrently started and may interleave each other. To ensure continuous operation throughout the runtime of the RTOS application, tasks often employ a *while(true)* loop, as illustrated in the example task depicted in Fig. 1a. A key responsibility of an RTOS is to schedule tasks, i.e., to decide which task should be executed at a given point of time. To manage tasks and their states, many real-time systems use a task-control block (TCB) model. It is one of the most popular methods to manage different numbers of tasks and is compatible with any specific scheduling strategy. The TCB model specifies that each task of a given real-time system is linked with a data structure called task-control block containing at least a program counter, an identifier, register contents, a status (or state), and a priority, if provided [22]. Most RTOS follow a task state scheme that is similar to the OSEK/VDX standard as shown in Fig. 2 [23]. Tasks typically start in a *suspended* state and become *ready* after activation. After system initialization, the task with the highest priority starts *running*. The running task may terminate its execution and become suspended again, it may wait for an event or a resource and become *waiting*, or it may be preempted by the scheduler if a task with a higher priority or some prioritized execution unit becomes ready. Tasks are released from



(a) Task

(b) Cyclic Handler

Fig. 1 Example Task and Cyclic Handler in EV3RT Style

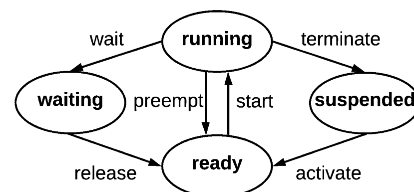


Fig. 2 Task States [23]

a *waiting* state if, for example, a resource becomes available or an event occurs. Note that other RTOS standards use slightly different terms, e.g., *blocked* instead of *waiting* in TOPPERS compatible systems or *pending* instead of *ready* in VxWorks.

Handlers and interrupt subroutines In addition to tasks, RTOS typically support prioritized execution units, e.g., interrupt subroutines or cyclic handlers. These prioritized execution units are similar to tasks but are typically not preemptable, i.e., they are executed from the beginning to the end whenever they become ready due to an external event (in case of interrupt subroutines) or the expiration of a periodic delay (in case of cyclic handlers). In this paper, we focus on cyclic handlers, which are executed periodically with a given period. Figure 1b shows a simplified example of a cyclic handler that reads a value from a sensor every 100 ms and starts at time 0.

Events In most embedded systems, tasks are either executed periodically (e.g., using cyclic handlers) or they are triggered by events. Most RTOS support events by providing mechanisms to *notify* an event, *wait* for an event and to *release* tasks that are waiting for an event. Often, bit patterns are used to wait for multiple events at the same time. Notifying tasks can then set bits that correspond to specific events within the bit pattern, while tasks that are waiting for one or more events define a corresponding bit mask on the bit pattern.

Scheduler The scheduler is a system program within a real-time system specifying the execution order of execution units. In this paper, we assume a single processor system, so only one execution unit can be executed at a time. Schedulers can implement different scheduling strategies, for example, round-robin, first-come-first-served (FCFS), preemptive-priority, or a mixture of these methods [22]. Most RTOS use preemptive and priority based scheduling.

Data queues Data queues are used in many custom RTOS for communication between tasks or with external applications (for example, via Bluetooth or WiFi). Data queues are kernel objects that are used for synchronization and communication by sending or receiving messages. Although data queues are only abstractly defined in typical RTOS standards such as OSEK/VDX or TOPPERS, we can generally distinguish between synchronous and asynchronous data queues. In synchronous data queues, the sender and receiver are synchronized, i.e., senders and receivers are blocked until a communication partner is available. To achieve this, synchronous data queues switch to a state where they only accept receivers whenever a task sent something to the queue, and only senders whenever a task tried to receive something, as

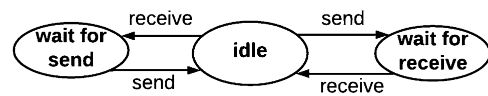


Fig. 3 Synchronous Data Queue

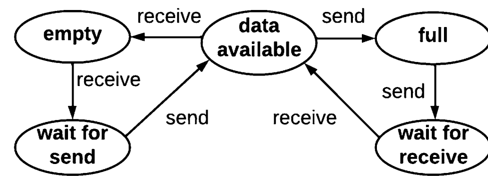


Fig. 4 Asynchronous Data Queue

depicted in Fig. 3. Asynchronous data queues can store a given number of messages (e.g., in a buffer), and send or receive calls are only blocked if the queue is empty or full, as shown in Fig. 4. In both types of data queues, requests that can not be accepted are usually stored in a send (resp. receive) queue and handled in first-in-first-out order.

2.2 UPPAAL timed automata

Timed Automata [4] are a timed extension of the classical finite state automata. A notion of time is introduced by real-valued clocks, which are used in clock constraints to model time-dependent behavior. Concurrent processes are modeled by networks of timed automata, which are executed with interleaving semantics and synchronize on channels. Formally, the semantics of timed automata and networks of timed automata are given by [8] as follows:

Definition 1 (Operational Semantics of a Timed Automaton)
A timed automaton (TA) is a tuple (L, l_0, C, A, E, I), where

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- C is a finite set of clock variables,
- A is a finite set of actions,
- $B(C)$ is a set of clock constraints with $x \sim n$ or $x - y \sim n$ for $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, and $n \in N$,
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges. We write $l \xrightarrow{a,g,r} l'$ for $(l, a, g, r, l') \in E$.
- $I : L \rightarrow B(C)$ assigns invariants to locations.

The semantics of a TA is defined as a transition system $(S, s_0, \rightarrow)$, where

- $S \subseteq L \times \mathbb{R}_{\geq 0}^{|C|}$ is the state space, where each state $s \in S$ is a pair of location and clock valuation (l, u). A clock valuation is a function $u : C \rightarrow \mathbb{R}_{\geq 0}$ that maps a nonnegative real value to each clock.
- $s_0 = (l_0, u_0)$ is the initial state consisting of an initial location l_0 and an initial clock valuation u_0.

– $\rightarrow$ is the transition relation with $\rightarrow \;\subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ specifying the conditions under which transitions occur.

A semantic step of a timed automaton can either be a time step (1) or a discrete transition (2) along an edge in the graphical representation:

(1) $(l,u) \xrightarrow{d} (l,u+d)$ iff $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$,
(2) $(l,u) \xrightarrow{a} (l',u')$ iff $l \xrightarrow{a,g,r} l'$ such that $u \in g \wedge u' = [r \mapsto 0]u \wedge u' \in I(l')$.

Here, $u \in I$ denotes that a clock valuation satisfies an invariant, and $u' = [r \mapsto 0]u$ denotes that all clocks from the clock set r are reset to zero.

Definition 2 (Semantics of a Network of Timed Automata) A network of timed automata (NTA) consists of n timed automata $\mathcal{A}_i = (L_i, l_{0,i}, C, A, E_i, I_i)$. The semantics of NTA is defined by a transition system $(S, s_0, \rightarrow)$, where

– $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^{|C|}_{\geq 0}$ is the state space, where each state $s \in S$ is a tuple $(\bar{l}, u)$; $\bar{l}$ is a location vector and u is a clock valuation.
– $s_0 = (\bar{l}_0, u_0)$ is the initial state consisting of a vector of initial locations $\bar{l}_0$ and an initial clock valuation u_0.
– $\rightarrow \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ is the transition relation.

A semantic step can be either a time step (1), an independent step of a single automaton (2), or a synchronization between two automata (3). Here, $c!, c? \in A$ represent input (resp. output) actions; $\tau \in A$ denotes an internal action and g denotes a clock guard; $I(\bar{l})$ denotes the conjunction of all invariants $I_i(l_i)$:

(1) $(\bar{l},u) \rightarrow (\bar{l},u+d)$ iff $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(\bar{l})$,
(2) $(\bar{l},u) \rightarrow (\bar{l}[l_i'/l_i],u')$ iff $l_i \xrightarrow{\tau g r} l_i'$ such that $u \in g \wedge u' = [r \mapsto 0]u \wedge u' \in I(\bar{l}[l_i'/l_i])$,
(3) $(\bar{l},u) \rightarrow (\bar{l}[l_j'/l_j, l_i'/l_i],u')$ iff $l_i \xrightarrow{c?g_i,r_i} l_i' \wedge l_j \xrightarrow{c!g_j,r_j} l_j'$ such that $u \in (g_i \wedge g_j) \wedge u' = [r_i \cup r_j \mapsto 0]u \wedge u' \in I(\bar{l}')$.

Uppaal [7–9] is a tool set for the modeling, simulation, animation, and verification of NTA. The Uppaal model checker enables the verification of temporal properties expressed in a subset of CTL. The simulator can be used to visualize counterexamples produced by the model checker. The Uppaal modeling language extends TA by introducing bounded integer variables, C-like functions, binary and broadcast channels, and urgent and committed location. TA are modeled as a set of locations, connected by edges. Invariants can be assigned to locations and enforce that the location is left before they would be violated. Edges may be labeled with selections, guards, synchronizations, and updates. Selections can be used to nondeterministically select a value from a given range. Updates are used to reset clocks and to manipulate the data space, where C is used as a host language. Processes synchronize by sending and receiving events through channels. Sending and receiving via a channel `c` is denoted by `c!` and `c?`, respectively. Binary channels are used to synchronize one sender with a single receiver. A synchronization pair is chosen nondeterministically if more than one is enabled. Broadcast channels are used to synchronize one sender with an arbitrary number of receivers. Urgent and committed locations are used to model locations where no time may pass. Leaving a committed location has priority over leaving uncommitted locations.

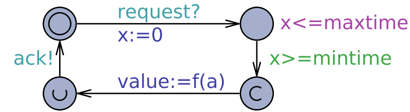


Fig. 5 Example Timed Automaton

An example Uppaal TA is shown in Fig. 5. The initial location is denoted by ◎and `request?` and `ack!` denote synchronizations on channels. The clock variable `x` is first set to zero and then used in two clock constraints: the invariant `x <= maxtime` denotes that the corresponding location must be left before `x` becomes greater than `maxtime`, and the guard `x >= mintime` enables the corresponding edge at `mintime`. The `value` is computed using a C function `f(a)`. The symbols ⓤ and © depict urgent and committed locations.

3 Reusable formal models for custom RTOS

Our key idea to reduce the effort of the formalization of systems that use a custom RTOS is to provide reusable abstract models for standard RTOS concepts and components. To achieve this, we combine the abstract formal model of RTOS components with a transformation for application level implementations of tasks and handlers. The resulting overall formal model can be used to analyze concurrency, synchronization, and timing behavior. Furthermore, it gives us access to existing analysis, verification, and simulation tools. Our overall approach is depicted in Fig. 6. The overarching goal is to analyze and verify *real time applications*. Those typically consist of (preemptable) *tasks* and (nonpreemptable) *handlers*. The *real time application* is executed by a *custom RTOS*, which provides a *scheduler* and manages *events, time* and *communication*. To ease the formalization of systems that use a custom RTOS, we provide reusable abstract TA models of core RTOS components. Two key components are a *configurable scheduler model* and a *generic task model*. The *configurable scheduler model* provides a general scheme to schedule preemptive and nonpreemptive execution units based on the task and handler information (e.g., identifier and priorities). It can be configured to a custom RTOS by implementing specific scheduling strategies. The *generic task*

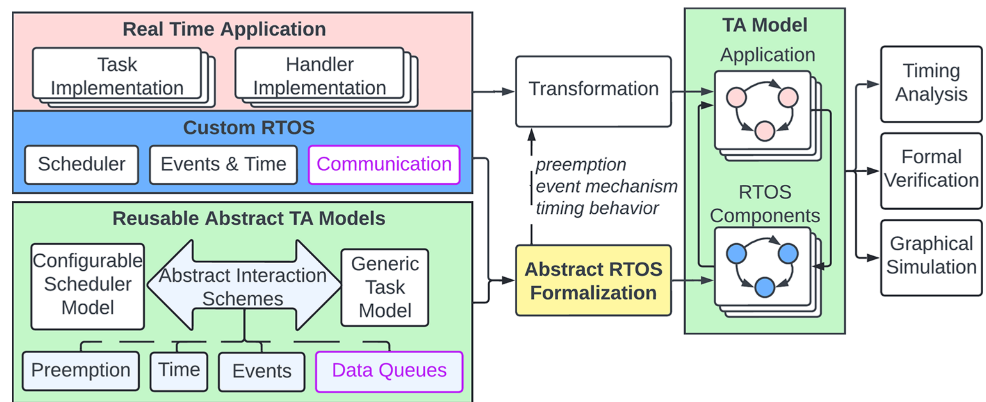


Fig. 6 Formalization with Reusable Abstract TA Models

model provides a generic model for the state of a task, which is compatible with most RTOS implementations, including OSEK, TOPPERS, VxWorks, and FreeRTOS compliant implementations. Our reusable models define *abstract interaction schemes* for the interplay between RTOS components and the application, such as *preemption*, *time*, *event* handling, and data queues for communication between tasks and with external communication partners, for example, via Bluetooth or WiFi. By customizing our reusable abstract TA models for a given custom RTOS implementation, an *abstract RTOS formalization* can be derived, which formally models central RTOS components like the scheduler. For the analysis of a given real-time application, the abstract formal RTOS model is then combined with a formalization of the application itself. The application code, i.e., the implementation of tasks and handlers, can potentially be automatically transformed into TA representations using existing transformations [19]. Our reusable formal models of *abstract interaction schemes* provide the necessary extensions for the interactions with the formal RTOS model, i.e., a preemption scheme, an event mechanism that enables us to transform system calls like wait functions or event notifications, and timing behavior that is, for example, implemented using sleep functions. The resulting TA model can be used for *timing analysis*, *formal verification* and *graphical simulation*. In particular, the Uppaal tool suite provides a powerful environment for graphical animation, simulation, model checking, and extensions for statistical model checking and test generation.

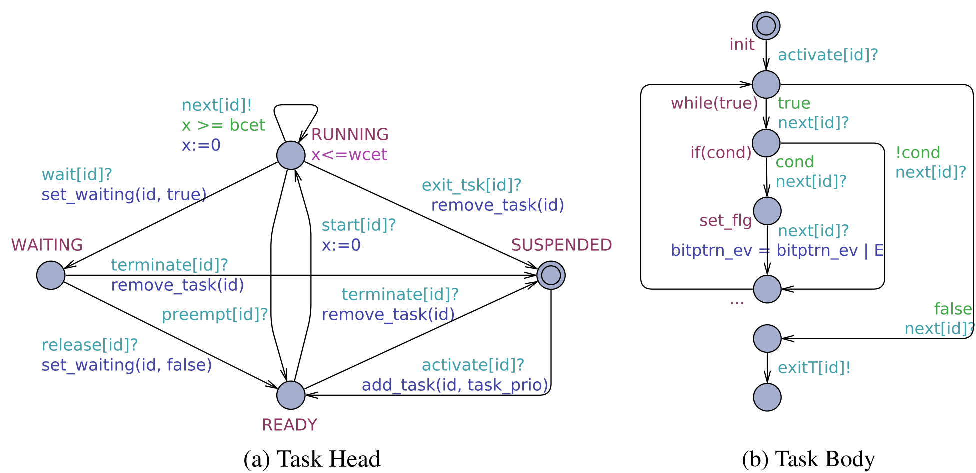
In the following, we present our reusable TA models for tasks, cyclic handlers, events and timing behavior, and a configurable scheduler model. We briefly discuss how sensor inputs are modeled. Finally, we present our reusable TA models for data queues, which can be used for internal and external communication.

3.1 Formalization of tasks

We model each task with two TA: one models the process states and controls the execution (*task head*), the other models the task implementation (*task body*). For the task head, we define a reusable TA template as shown in Fig. 7a. One location models each task state. We switch between these locations using synchronizations on channels. These channels are parameterized with the task *id*. This means that the task template needs to be included into a model only once and can be instantiated for all tasks. The task starts in the initial location *SUSPENDED*. It may be *activated* by the system initialization (which is also modeled as an automaton) at start-up or by other tasks. If a task is activated, its *id* and priority are inserted into the scheduler queue (*add_task()*). If the scheduler *starts* a task, a local clock x is reset to zero to model its execution time. A task body captures the code run by a task. While multiple tasks run concurrently and can interleave each other, each individual task body contains sequentially executed code statements. Consequently, we assume that the function executed within the task can be transformed into an equivalent TA using existing method transformation techniques [19]. The main idea can be summarized as follows: For every program statement in the task code, a corresponding transition is inserted in the task body TA. Assignments are added as updates. Conditional statements result in branching transitions, where the conditions are used as guards.

Certain Real-Time Operating System (RTOS) functionalities, such as synchronization through event flags, data acquisition from sensors, or intertask communication via data queues, necessitate specific modeling techniques. To accommodate these functionalities within the *task body*, we have established specialized TA templates. For instance, sensor data reading operations can use to the template detailed in Sect. 3.5, and event-based communications can use the design outlined in Sect. 3.3. Figure 7b shows an exemplary task body for the code given in Fig. 1a. The task runs a *while(true)* loop in which a condition is checked. If the condition holds, it sets an event flag to a bit pattern (see Sect. 3.3).

In the *RUNNING* state, we continuously trigger statements of the task body with a *next* synchronization. The *next* synchronization controls the execution of the task body: within



(a) Task Head (b) Task Body

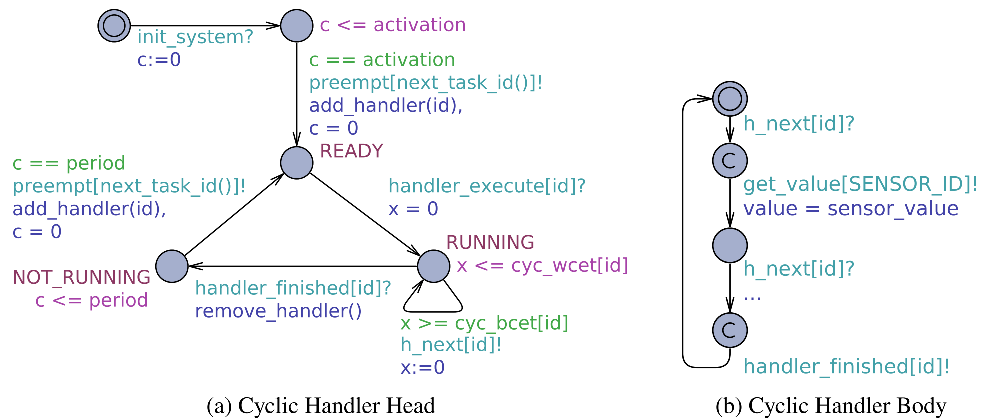
Fig. 7 Generic Task Model with Example Task Body

the body, each program statement is guarded with this synchronization, such that the next statement can only be executed if the task is still in its *RUNNING* state. Additionally, we use the *next* synchronization to model real-time behavior. The next statement can only be executed if at least the best case execution time (*bcet*) has expired and must be executed at the latest when the worst-case execution time (*wcet*) elapses. The *bcet* and *wcet* can be provided for each task individually as a global overapproximation of the execution times per statement of that task, or tailored to each statement by manipulating the global variables *bcet* and *wcet* at runtime. For a custom RTOS, we expect that these values are provided by the developer. The execution of the task can be terminated by the task itself by calling a termination function, modeled as a synchronization on *exitT*. It is then removed from the scheduler queue and becomes *SUSPENDED*. If a task waits for an event, a resource, or a given amount of time by using some kind of wait or sleep function, the task switches from *RUNNING* to *WAITING*, and the task is set to waiting in its task control block. The task becomes *READY* again if it is *released* by the corresponding event, resource, or if the time expires. Finally, task preemption is modeled by a synchronization *preempt*, which is used by the scheduler to switch the currently active task from *RUNNING* to *READY* if a task with a higher priority becomes ready for execution. External events or cyclic handlers may also preempt the currently running task, as they have precedence over all tasks. Note that if a task is preempted, the execution time x of the current statement is reset to zero the next time the task enters the running state. Thus, in this approach we assume that the currently executed program statement is terminated and all progress on the statement is lost.

By providing general formalizations for typical task states, best and worst case execution times, and preemption, our task templates are reusable for a large number of custom RTOS, including all that are OSEK/VDX compatible.

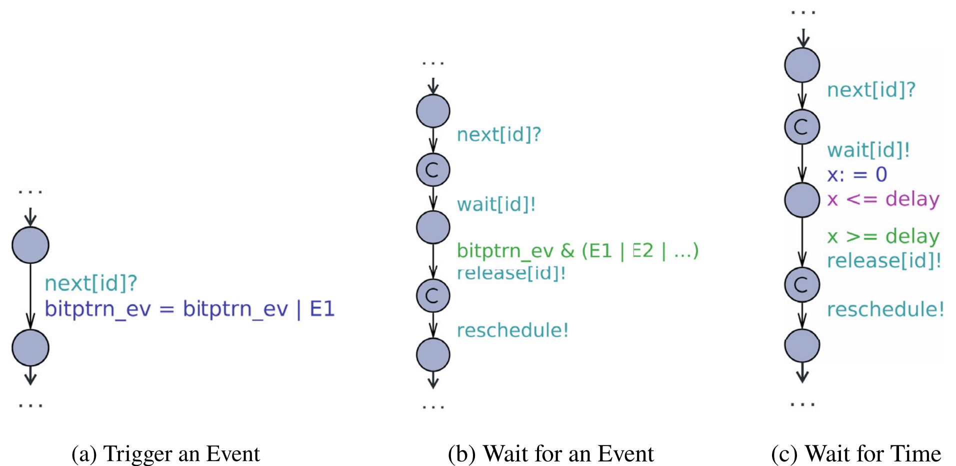
3.2 Formalization of cyclic handlers

As an illustrative example for the formalization of nonpreemptable, prioritized handlers, we define reusable TA templates for cyclic handlers. Similar to tasks, we model cyclic handlers with two TA: a reusable TA template for modeling states and controlling the execution (*cyclic handler head*), and a second automaton that captures the implementation (*cyclic handler body*). Figure 8a shows the cyclic handler head template. Cyclic handlers are activated with system initialization (*init_system*). They may have an initial offset (*activation*), which delays their first execution. To model offset and periodicity, we use a clock variable c. After initialization, the cyclic handler waits until c reaches its activation time. Then it switches into the *READY* state. With this transition, its *id* is inserted into the scheduler queue (*add_handler()*). Because cyclic handlers have higher precedence than aperiodic tasks, an activation directly leads to the preemption of the currently running task (*preempt*). At the transition to the *READY* state, c is reset. In the *READY* state, the cyclic handler waits to be executed by the scheduler (*handler_execute*), and then switches to *RUNNING*. In the *RUNNING* state, the process defined in the cyclic handler body is executed. Real-time behavior is modeled similar to tasks by synchronizing the execution of program statements with an *h_next* channel and an overapproximation of the execution times per statement with global worst-case (*h_wcet*) and best-case (*h_bcet*) execution times. Similar to the execution times of tasks, *h_wcet* and *h_bcet* can be manipulated at runtime for each handler. When the execution of the body finishes (*handler_finished*), the handler is removed from the scheduler queue (*remove_handler()*), and the state switches to the *NOT_RUNNING* state. When the *period* of the cyclic handler expires, the cyclic handler switches back to the *READY* state, c is reset, the handler is again added to the scheduler queue (*add_handler()*), and the currently running task is *preempted*.



(a) Cyclic Handler Head (b) Cyclic Handler Body

Fig. 8 Cyclic Handler Templates



(a) Trigger an Event (b) Wait for an Event (c) Wait for Time

Fig. 9 Events and Timing Behavior

The formalization of the cyclic handler body matches that of a task, as depicted in Fig. 8b, and can be derived similarly. The example corresponds to the example cyclic handler in Fig. 1b. Similar to tasks, every program statement in the cyclic handler body synchronizes with the cyclic handler head using the *h_next* channel. In this example, the cyclic handler body periodically reads a value from a sensor (see also Sect. 3.5). Both the scheduler and handler head are informed about completion of the body's execution by a synchronization on the broadcast channel *handler_finished*. This synchronization occurs immediately because the last location of a cyclic handler body is committed.

3.3 Events and timing behavior

To support events, RTOS typically provide functions to set or notify events and to wait for events. We propose to translate these functions together with the task or handler body where they are called into the necessary updates and synchronizations as illustrated in Fig. 9. Triggering an event usually involves setting some bit pattern, as shown in Fig. 9a. Tasks that wait for an event call a wait program statement, which leads to them switch into *WAITING* state; synchronization occurs. For this, they synchronize on a broadcast channel *wait*, which is sent from a committed location, as shown in Fig. 9b. The *wait* signal is also sent to the scheduler to trigger rescheduling, as the currently running task is now blocked. To ensure that rescheduling is performed with priority, we send signals that trigger rescheduling from a committed location. Then, the task waits for a bit pattern (representing the occurrence of one or more events) specified with the wait function call. If the relevant events are set in the bit pattern, the task synchronizes on the urgent channel *release*, which results in switching from *WAITING* to *READY*. If the released task has a higher priority than the currently running task, the scheduler is informed that it might need to *reschedule* via an urgent broadcast channel, which ensures that no time may pass. If the scheduler misses the signal *reschedule*, it is currently not running a task and will reschedule anyway.

The wait-release mechanism can also be used to model timing behavior. Most RTOS provide some kind of sleep function, which take a timing *delay* as a parameter and switch the calling task to the *WAITING* state for the given amount of time. We can transform these kind of functions into a timed wait as shown in Fig. 9c. It uses the same sequence of *wait*, *release*, and *reschedule*, but now does not wait for an event

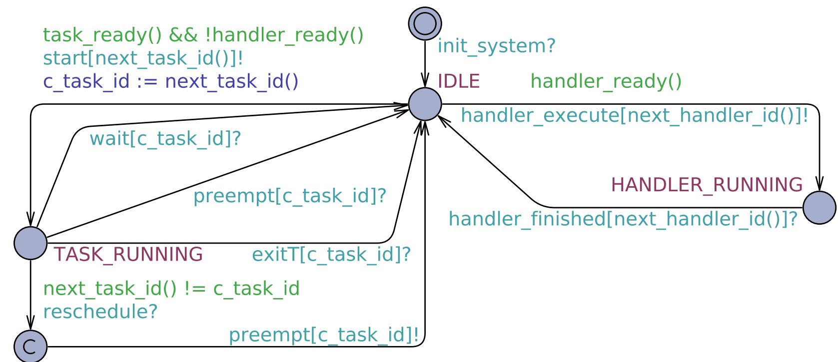


Fig. 10 Configurable Scheduler Model

but for the given *delay* by setting a local clock x to zero and then waiting until x is equal to the given *delay*.

Fig. 11 Abstract Sensor Model

3.4 Configurable formal scheduler model

Figure 10 shows our TA template for a configurable scheduler. It is reusable for all custom RTOS that support separate execution modes for tasks and prioritized nonpreemptive execution units such as interrupt subroutine or cyclic handlers. The scheduler manages task and handler information in separate queues, which can be used to determine whether some handler or task is ready for execution (*handler_ready(), task_ready()*), and to determine the handler or task with the highest priority (*next_handler_id(), next_task_id()*). The functions *next_handler_id()* and *next_task_id()* can be used to implement custom scheduling strategies that are compliant with the overall preemptive, priority-based execution scheme.

After system initialization, the scheduler first checks whether a handler is ready for execution. If this is the case, the handler with the highest priority (*next_handler_id()*) is activated by synchronizing on *handler_execute*. As handlers may not be preempted, the scheduler then just waits for the handler to signal termination via the *handler_finished* channel. If further handlers are ready for execution, they are then executed in the order of precedence provided by *next_handler_id()*. Only if no handler is ready anymore, tasks are executed. To this end, the scheduler activates the task with the highest priority via the channel *start* and stores its task id in the local variable *c_task_id*. While the task is running, it may voluntarily give up control by termination (*exitT*) or by waiting for an event or time (*wait*). In addition, it may be *preempted* by a handler becoming ready for execution, or by tasks with a higher priority. To capture the latter, we synchronize on *reschedule* whenever the task with the highest priority has changed. This may happen due to tasks waking up after a timed delay or due to an event notification. If this happens, we *preempt* the currently running task and reschedule. We use a committed location to ensure that preemption, which is considered to be essential for rescheduling, has priority over other enabled transitions (such that *reschedule* and *preempt* are always executed together).

3.5 Modeling sensor inputs

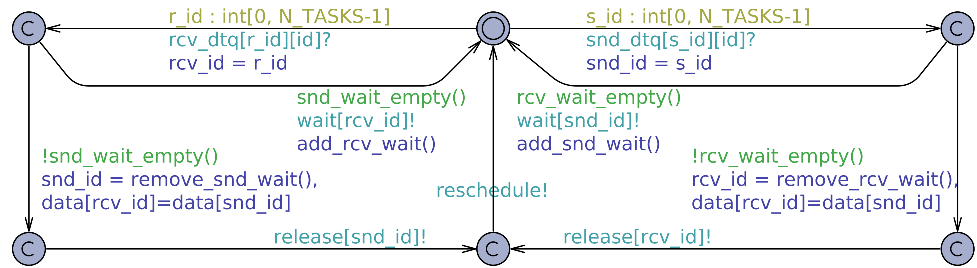
To generate new sensor inputs, we propose the reusable TA model in Fig. 11. The template consists of a single location with a self-loop transition. The template consists of a single location with a self-loop transition. If a process synchronizes via the *get_value* channel with the *SENSOR_ID* the variable for the *sensor_value* is updated with a nondeterministically chosen value from the sensor range (using a nondeterministic *selection* in Uppaal).

Processes executed in task or cyclic handler bodies can read in new sensor values during execution by synchronizing on the *get_value* channel and assigning the current *sensor_value* to a local variable *value*, as shown, for example, in the body of the cyclic handler in Fig. 8. This *sensor_value* synchronization is triggered immediately from a committed location as soon as the program statement is finished via the *h_next* synchronization.

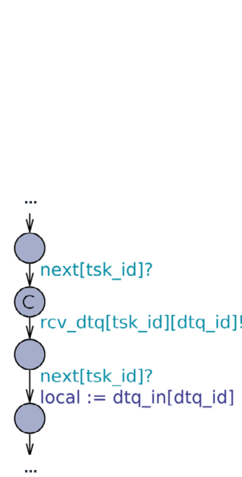
3.6 Communication with data queues

To integrate communication via data queues into our approach, we provide additional timed automata templates for them. To model communication with data queues, it is important that the tasks that perform send and receive operations are properly synchronized with the scheduler. A data queue can block tasks until a communication partner is available for synchronous communication. For asynchronous communication, a data queue can block a sending task if the data buffer of the queue is full, or block a receiving task if the buffer is empty. Our data queue templates can block a sending

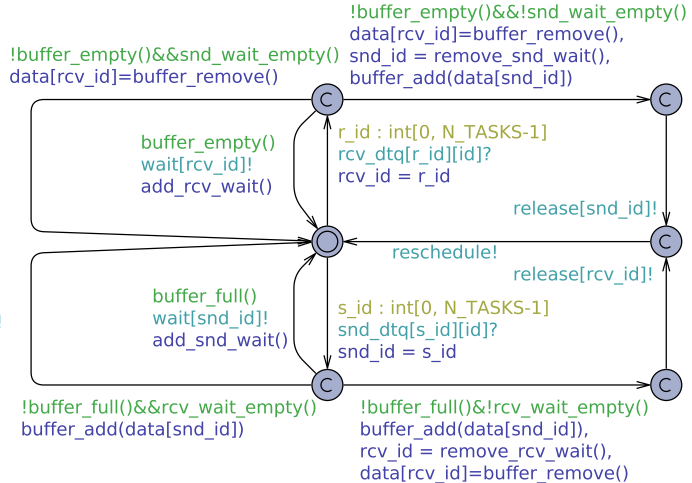
Fig. 12 Data Queue Templates



(a) Reusable Timed Automata Model for Synchronous Data Queues



(b) Receive from a Data Queue



(c) Reusable Timed Automata Model for Asynchronous Data Queues

or a receiving task by synchronizing on the *wait* and *release* channels of the task head automaton associated with the corresponding task. The resulting timed automata templates that abstractly model communication through data queues are shown in Figs. 12a and 12c. Both templates can be configured with the following parameters:

– a unique identifier *id*,
– a pointer to a data structure *data* that allows us to store send and receive requests (with one entry per task),
– a maximum number of senders or receivers that can be queued as pending send or receive requests.

The asynchronous data queue is additionally parameterized with the size of the buffer used for communication.

Figure 12a illustrates our reusable timed automata template for synchronous data queues, which can handle send and receive requests. For universal synchronization across tasks for every data queue, we utilize two-dimensional channel arrays: *rcv_dtq[r_id][id]* and *snd_dtq[s_id][id]*. Here, the first index signifies the ID of either a receiving (*r_id*) or a sending (*s_id*) task, while the second points to the data queues ID. In its initial state, the synchronous data queue is ready to send or receive from all tasks. The non-deterministic selection of *r_id* and *s_id* guarantee synchronization with any task within the specified ID range.

If a receive request is received, the receiver id (*r_id*) is stored into a local variable *rcv_id* and the data queue switches into a committed location. For synchronous communication, we then check whether a sender is already waiting, and distinguish the following two cases:

1. If no sender is waiting (*snd_wait_empty()*), the receiver is blocked by synchronizing on the *wait* channel of the corresponding task head automaton. The task head will then switch into the *WAITING* state and will effectively block the task by not sending *next* anymore. The *wait* is also received by the scheduler to inform it that no task is running anymore, and that another task can be scheduled. Furthermore, the receiving task is added to a FIFO queue (*add_rcv_wait()*) to ensure that subsequent receive requests are managed in FIFO order. The implementation of the FIFO queue can be overwritten for other queuing schemes.
2. If at least one sender is already waiting for communication (*!snd_wait_empty()*), a sender *id* is dequeued from the FIFO queue managing pending send requests (*snd_id = remove_snd_wait()*, the data exchange is executed (*data[rcv_id]=data[snd_id]*), and the sending task is released using a synchronization on the *release* channel of its task head.

Send requests are handled analogously.

Figure 12c depicts our reusable timed automata template for asynchronous data queues. The asynchronous data queue implements a ring buffer that is used to store a given maximum number of data packets. Synchronization with send and receive tasks is analogous to the synchronous data queue. If the asynchronous data queue receives a receive request (via *rcv_dtq[r_id][id]*), the *r_id* of the receiver is again stored in a local variable *rcv_id*. Then, we switch into a committed location, where we distinguish the following three cases:

1. If the buffer is empty (*buffer_empty()*), the receiver is blocked by synchronizing on the *wait* channel of the corresponding task head automaton, and the task id of the receiver is enqueued in the FIFO queue managing receive requests (*add_rcv_wait()*).
2. If the buffer is not empty and no sender is waiting (*!buffer_empty() && snd_wait_empty()*), the receiver can read data from the internal ring buffer (*data[rcv_id]=buffer_remove()*).
3. If the buffer is not empty and a sender is waiting (*!buffer_empty() && !snd_wait_empty()*), the receiver can read data from the internal ring buffer and additionally, a pending sender (taken from the FIFO queue managing send requests, *snd_id = remove_snd_wait()*) can write its data into the now free place in the buffer. The previously blocked sender is unblocked using a synchronization on the *release* channel of its task head.

Send requests are handled analogously.

Figure 12b displays how a task or cyclic handler can receive data from a data queue. After the *next* synchronization, the *rcv_dtq* channel synchronizes with the data queue. If the queue is empty, sending a receive request can put the task into *WAITING* state. If the task is *RUNNING* again, data can be read from the *dtq_in* array with the following *next* synchronization. Sending to a data queue works analogously.

The two timed automata templates shown in Figs. 12a and 12c abstractly capture synchronous and asynchronous communication and take care of the synchronization between data queues, tasks, and the scheduler. They can be configured for different packet types and management schemes for pending requests. With that, they are highly reusable for a broad variety of custom RTOS. We discuss their application to formalize the custom RTOS EV3RT in the next section.

4 Case study: search and rescue robots

To evaluate our approach, we have used our reusable formal models to abstractly capture the execution semantics of the custom RTOS EV3RT and manually translated an application that implements a search and rescue robot for LEGO Mindstorms into UPPAAL.

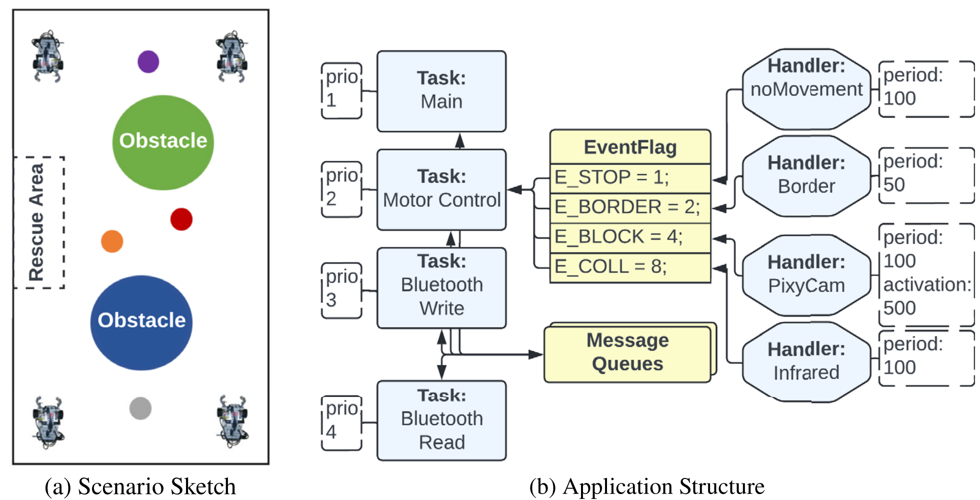
EV3RT is a real-time operating system based software platform to develop real-time applications for the LEGO® Mindstorms EV3. It uses the TOPPERS/HRP2 kernel, which provides features such as preemptive multitasking and memory protection [14]. The kernel also offers kernel objects, such as tasks, cyclic handlers, eventflags and data queues to simplify the development of real-time applications [1].

The search and rescue robot application was developed as part of a student's project. Multiple EV3 robots are tasked with locating and evacuating objects out of a danger zone and into a rescue area. A server assigns search and evacuation routes to the robots. The robots communicate with the server via Bluetooth, detect obstacles with an infrared sensor, the border of the experiment table with a color sensor, and objects that should be evacuated with a pixy camera. The structure of the EV3RT application is shown in Fig. 13b. Each robot program consists of four tasks: a global main task, a motor control task, and two tasks for reading and writing Bluetooth packages. Note that lower *prio* values indicate higher priorities. Three cyclic handlers periodically read from the color and infrared sensors and the pixy cam. Another cyclic handler periodically checks whether the robot is moving. An event flag is used to alert the motor control task to several events, namely that the robot stopped, that it has reached the map border, or that the infrared sensor has detected a possible imminent collision.

With our reusable formal models, it was straight-forward to manually translate the rescue robot application into UPPAAL timed automata. The resulting model is available at https://github.com/EmbSys-WWU.

To construct our formal model, we instantiated templates for the necessary kernel objects, i.e., events, data queues, task heads, and cyclic handler heads, according to the architecture shown in Fig. 13b. In line with the EV3RT implementation, we have configured our scheduler model such that only the task with the highest priority is executed and an FCFS strategy is chosen for equal priorities. We have transformed task and cyclic handler bodies manually to UPPAAL TA following the process described in Sect. 3.1 and using our reusable formalization of activation, preemption, and wait function calls by mapping the general concepts to the EV3RT-specific system calls, e.g., `act_tsk()` to the channel *activate*, `exit_tsk()` to the channel *exitT*, `set_flg()` to a transition that sets the bit pattern of the given event, `wai_flg()` to the wait-release mechanism, `clr_flg()` to a transition that resets the bit pattern, and `ev3_infrared_sensor_get_distance()` to *get_value[IR_SENSOR]*. In a future work, we plan to automate this process by combining our approach with existing method transformation techniques [19]. We have modeled sensor inputs with the sensor template shown in Fig. 11. We have used our reusable data queue models to model both

Fig. 13 Search and Rescue Robot Application



(a) Scenario Sketch (b) Application Structure

synchronous and asynchronous communication. In particular, in our case study the incoming Bluetooth messages are received by a Bluetooth Read Task and then synchronously communicated to the Main Task, while outgoing Bluetooth messages are asynchronously communicated to the Bluetooth Write task (which forwards them to the server). Note that we have abstracted from the server by manually defining a TA that generates Bluetooth messages. We have also abstracted from most functional variables, i.e., the position of the robot, the navigation and PID controller, and the specific sensor values. For the latter, we only distinguish two values for each sensor (i.e., obstacle/border/object detected or not detected). In summary, the resulting formal model abstracts from data, but precisely captures concurrency, synchronizations, timing, communication, and the reaction to external events. These are typically particularly hard to test and debug, while many errors arise from faulty task integration, misunderstandings of the scheduling semantics, and timing issues.

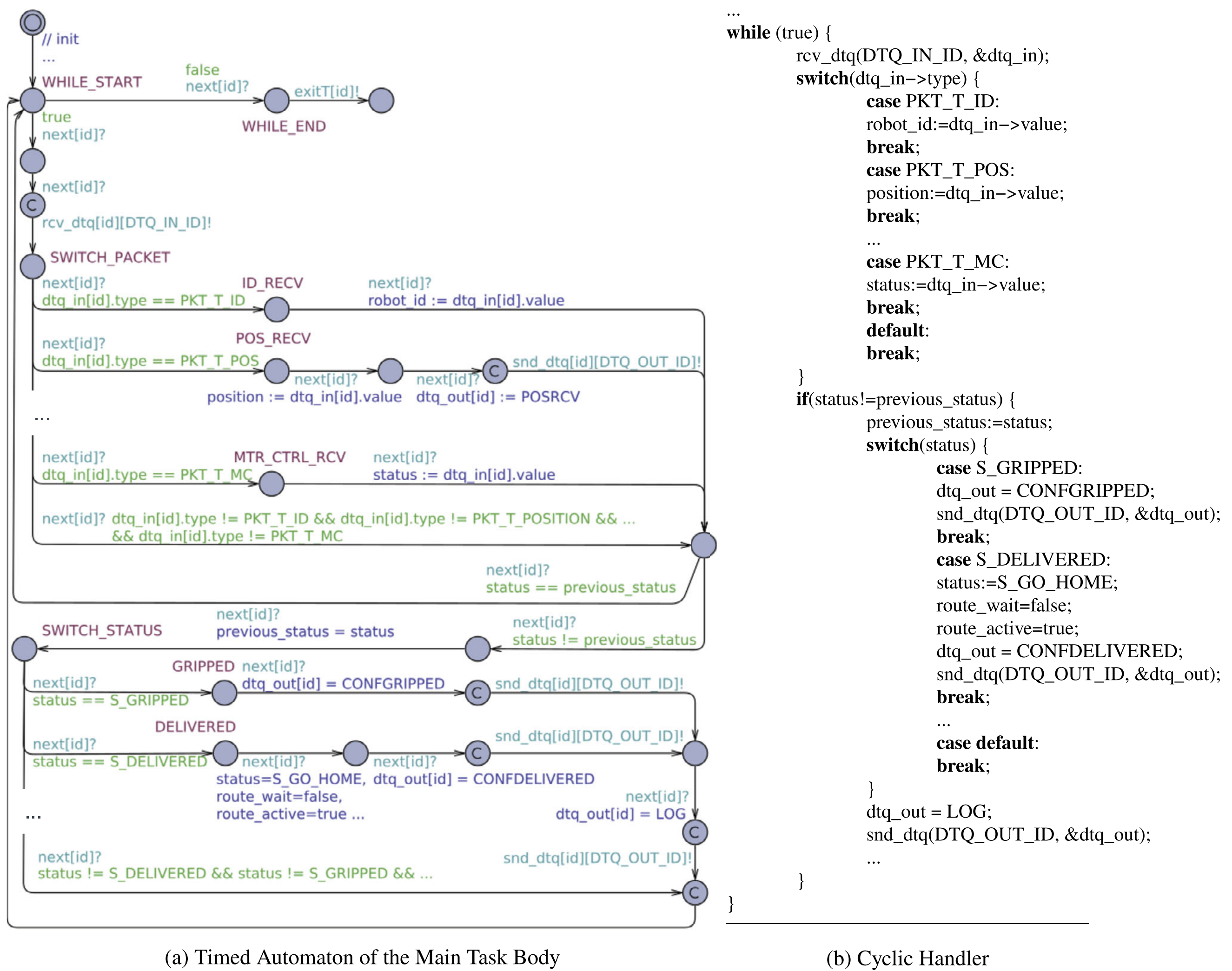
With the formal TA model, we were able to simulate possible sequences of events and actions, without the necessity to execute the software on the real hardware, which is very error-prone and time consuming. In contrast to the real execution, which is not only slow but also very difficult to debug, we were able to manually trace possible executions and interleavings between tasks and handlers, with timing and state information. The graphical animation proved to be extremely helpful for this manual validation process.

To illustrate the interaction of tasks with data queues (and the scheduler), Fig. 14a shows an excerpt of the body of the *Main* task. The corresponding EV3RT code is shown in Fig. 14b. The *Main* task executes a control loop where it handles packets from both the server and the motor control task. If new commands are received from the server, it stores the new information, for example, a new ID, a new position, a new path, or a new goal. The motor control task informs the main task whenever its status has changed, for example, if it has gripped or delivered a rescue object.

The timed automaton shown in Fig. 14a starts with some initializations and then goes into a *while(true)* loop, which serves as the main control loop. Within the loop, it receives packets from the synchronous input data queue *DTQ_IN_ID*. To properly synchronize the task with both the data queue and the scheduler, we first synchronize on *next[id]?*. As explained above, this is only possible if the task is currently running and executed by the scheduler. Then, we use a committed location to send the receive request to the data queue, i.e., we synchronize on *rcv_dtq[id][DTQ_IN_ID]*. Our synchronous data queue model will put the task head into its *WAITING* state if no communication partner is available yet, and *release* it whenever a sender is available. Together, this ensures that the main task will only continue execution if the communication has happened and the scheduler has chosen this task for execution again.

If the *Main* task received a packet, it handles them in two subsequent switch statements: First, it switches over the type of the packet. For some commands, for example, if a new ID, position, or path is received, it just reads the value of the packet *dtq_in[id].value* and stores it into appropriate variables, for example, *robot_id* or *position*. For other commands, for example, the status updates received from the motor control task, it changes the *status* variable, which is used in the second switch statement. The second switch statement switches over the new *status*, e.g., *GRIPPED* or *DELIVERED*, and sends a confirmation packet to the server by sending it to the asynchronous output data queue *DTQ_OUT_ID*.

Sending a packet through asynchronous output data queue is analogous to receiving a packet: The task first synchronizes on *next[id]?* and then on *snd_dtq[id][DTQ_OUT_ID]*. Our timed automata model of an asynchronous data queue ensures that the task will only be blocked at this point if the queue is full.




```
...
while (true) {
    rcv_dtq(DTQ_IN_ID, &dtq_in);
    switch(dtq_in−>type) {
        case PKT_T_ID:
        robot_id:=dtq_in−>value;
        break;
        case PKT_T_POS:
        position:=dtq_in−>value;
        break;
        ...
        case PKT_T_MC:
        status:=dtq_in−>value;
        break;
        default:
        break;
    }
    if(status!=previous_status) {
        previous_status:=status;
        switch(status) {
            case S_GRIPPED:
            dtq_out = CONFGRIPPED;
            snd_dtq(DTQ_OUT_ID, &dtq_out);
            break;
            case S_DELIVERED:
            status:=S_GO_HOME;
            route_wait=false;
            route_active=true;
            dtq_out = CONFDELIVERED;
            snd_dtq(DTQ_OUT_ID, &dtq_out);
            break;
            ...
            case default:
            break;
        }
        dtq_out = LOG;
        snd_dtq(DTQ_OUT_ID, &dtq_out);
        ...
    }
}
```

(a) Timed Automaton of the Main Task Body (b) Cyclic Handler

Fig. 14 Transformation of the Main Task Body

5 Experimental results

We have used the Uppaal model checker to verify the reachability properties (1)–(6) shown in Table 1, namely that the possible sequences of events and actions include sequences where the main tasks reaches locations, corresponding to mission states in which the robot receives a goal, moves to the goal, grips an object, delivers it to the rescue zone, completes the delivery. Note that not all locations corresponding to the queries are shown in Fig. 14a for reasons of brevity. Additionally, we check that there exists paths in which the robot returns to its home position, indicated by the status *S_HOME*. All of these reachability properties were verified using a random depth first search.

Although our formal model abstracts from some data variables, it contains the process interactions and their (approximate) timing, as well as nondeterministic overapproximations for sensor inputs and communication via data queues. As a result, the number of potential states expands considerably, even for basic applications. However, our approach is well suited to verify properties for fixed scenarios. To validate our formal model of the scheduler and process and handler interactions, we limit our model to a fixed mission sequence that ensures mission success, detailed as follows:

– We generate one specific sequence of commands that are received via Bluetooth from the server.
– The sensor values are restricted to values with which the mission succeeds, i.e., the pixy Cam always spots the target, and neither collision occur nor is the border ever reached.

In [3], we fixed the worst-case execution time (*wcet*) and best-case execution time (*bcet*) uniformly to one. However, subsequent improvements to our model have led to consider-

Table 1 Verified properties for the search and rescue robots

	Reachability Properties
(1)	$E\Diamond\ mainTaskBody.\texttt{GOAL_RECV}$
(2)	$E\Diamond\ mainTaskBody.\texttt{GET_GOAL}$
(3)	$E\Diamond\ mainTaskBody.\texttt{GRIPPED}$
(4)	$E\Diamond\ mainTaskBody.\texttt{GO_DELIVER}$
(5)	$E\Diamond\ mainTaskBody.\texttt{DELIVERED}$
(6)	$E\Diamond\ status = \texttt{S_HOME}$
	Safety Properties (Fixed Mission Progress)
(7)	$A\Box\neg deadlock$
(8)	$A\Box\ (\forall\ t_i \in Tasks : t_i.\texttt{RUNNING} \rightarrow \neg(\exists\ t_j \in Tasks : t_i.id \neq t_j.id \wedge t_j.\texttt{RUNNING}))$
(9)	$A\Box\ (\forall\ t_i, t_j \in Tasks : t_i.\texttt{RUNNING} \wedge t_j.\texttt{READY} \wedge t_j.prio < t_i.prio \rightarrow t_j.c_{\texttt{READY}} \leq 0)$
(10)	$A\Box\ (\forall\ t_i \in Tasks : t_i.\texttt{RUNNING} \rightarrow \neg(\exists\ h_j \in Handlers : h_j.\texttt{READY} \vee h_j.\texttt{RUNNING}))$
(11)	$A\Box\ (\forall\ h_i \in Handlers : h_i.\texttt{RUNNING} \rightarrow h_i.c \leq h_i.period)$
(12)	$A\Diamond\ Tasks(\texttt{MAIN_ID}).\texttt{RUNNING}$
(13)	$A\Diamond\ Tasks(\texttt{MOTOR_ID}).\texttt{RUNNING}$
(14)	$A\Diamond\ Tasks(\texttt{BTREAD_ID}).\texttt{RUNNING}$
(15)	$A\Diamond\ Tasks(\texttt{BTWRITE_ID}).\texttt{RUNNING}$
	DTQ Properties (Fixed Mission Progress)
(16)	$A\Box\ \neg(status = \texttt{S_HOME}) \wedge mainTaskBody.\texttt{RCV_REQ} \rightarrow A\Diamond\ mainTaskBody.\texttt{WHILE_START}$
(17)	$A\Box\ \neg(status = \texttt{S_HOME}) \wedge \neg DTQin.\texttt{rcv_wait_empty}() \rightarrow A\Diamond\ DTQin.\texttt{rcv_wait_empty}()$
(18)	$A\Box\ \neg DTQin.\texttt{rcv_wait_empty}() \rightarrow Tasks(\texttt{MAIN_ID}).\texttt{WAITING}$
(19)	$A\Box\ \neg DTQout.\texttt{rcv_wait_empty}() \rightarrow A\Diamond\ DTQin.\texttt{rcv_wait_empty}()$
(20)	$A\Diamond\ \neg DTQout.\texttt{buffer_empty}()$
	Wait Time Properties (Fixed Mission Progress)
(21)	$A\Box\ \neg(status = \texttt{S_HOME}) \wedge Tasks(\texttt{MAIN_ID}).\texttt{WAITING} \rightarrow Tasks(\texttt{MAIN_ID}).c_{\texttt{WAITING}} \leq 2000$
(22)	$A\Box\ \neg(status = \texttt{S_HOME}) \wedge Tasks(\texttt{MOTOR_ID}).\texttt{WAITING} \rightarrow Tasks(\texttt{MOTOR_ID}).c_{\texttt{WAITING}} \leq 120$
(23)	$A\Box\ \neg(status = \texttt{S_HOME}) \wedge Tasks(\texttt{BTWRITE_ID}).\texttt{WAITING} \rightarrow Tasks(\texttt{BTWRITE_ID}).c_{\texttt{WAITING}} \leq 4000$
(24)	$A\Box\ \neg(status = \texttt{S_HOME}) \wedge Tasks(\texttt{BTREAD_ID}).\texttt{WAITING} \rightarrow Tasks(\texttt{BTREAD_ID}).c_{\texttt{WAITING}} \leq 2000$
	DTQ Properties (High Buffer Pressure & Fixed Mission Progress)
(25)	$E\Diamond\ DTQout.\texttt{buffer_full}()$
(26)	$A\Box\ DTQout.\texttt{buffer_full}() \rightarrow A\Diamond\ \neg DTQout.\texttt{buffer_full}())$

able optimizations. Notably we exchanged the data structures managing task and handler waiting queues by implementing ring buffers and replaced a default channel in Bluetooth message generation that could lead to infinite wait times with an urgent channel, which sends a message as soon as possible. With these improvements, we were able to verify all properties for *wcet*=2 and *bcet*=1. Note that we assume that the values for the *bcet* and *wcet* are provided by the developer and that the values used in this cases study serve validation purposes only. For the fixed mission sequence model, we have verified the safety properties (7)–(15) in Table 1 using Uppaal default settings. Here, the *Tasks* variable refers to an array comprising *Task Head* automata, each accessible through their respective task ids. The queries show that the scheduler behaves as intended, i.e., that it never deadlocks (7), that only one task is running at a time (8), that tasks are executed according to their priorities (9), that only one handler is running at a time (10), that cyclic handlers are always executed within the specified cycle (11). Properties (12)–(15) show that all tasks are executed.

In addition to the properties we have presented in [3], we have also verified various properties that validate correct synchronization with our data queue models. We again use our model with the fixed mission sequence for this.

To ensure that the main task receives data from the synchronous input data queue correctly, we have verified properties (16)–(18). Property (16) states that whenever the *Main* task starts a receive request in the *RCV_REQ* location, it eventually receives something, i.e., it will return to the *WHILE_START* location at some point. Property (17)

Table 2 Verification times and memory requirements and explored states for the properties in Table 1

	bcet = *wcet*		*bcet* < *wcet*		Increase factor	
	hh:mm:ss:ms	Explored states	hh:mm:ss:ms	Explored states	Incr. time	Incr. states
(1)	10	2072	1	2060	0.10	0.99
(2)	20	2553	20	2608	1.00	1.02
(3)	10	2844	20	3158	2.00	1.11
(4)	20	3134	20	3198	1.00	1.02
(5)	30	3744	20	4332	0.67	1.16
(6)	20	4027	30	4040	1.50	1.00
(7)	5:28:010	375748	44:24:650	50570956	8.12	134.59
(8)	5:08:940	386796	39:40:090	50570956	7.70	130.74
(9)	7:50:610	425772	1:33:40:120	65675056	11.94	154.25
(10)	5:07:240	425772	39:37:960	50570956	7.74	118.77
(11)	5:05:550	425772	40:05:200	50570956	7.87	118.77
(12)	790	129	5:020	129	6.35	1.00
(13)	10	339	1	339	0.10	1.00
(14)	1	441	10	441	10.00	1.00
(15)	10	495	0	495	0.00	1.00
(16)	5:14:140	4837738	2:40:09:960	129637594	30.59	26.80
(17)	5:12:780	4804798	2:58:44:270	127304106	34.29	26.50
(18)	5:07:420	4197514	39:35:020	50570956	7.73	12.05
(19)	19:51:640	10916938	5:18:53:660	176434047	16.06	16.16
(20)	800	309	5:160	309	6.45	1.00
(21)	6:42:280	4371634	3:42:31:920	89762878	33.19	20.53
(22)	6:08:510	4197514	54:02:330	52707934	8.80	12.56
(23)	6:37:250	4227550	7:13:33:470	109532584	65.48	25.91
(24)	6:11:330	4201246	54:55:960	51766048	8.88	12.32
(25)	10	683	10	3308	1.00	4.84
(26)	4:690	670792	1:24:44:090	96438330	1084.03	143.77

states that whenever there is a receiving task (the *Main* task) waiting to be served by the synchronous input data queue, the task will be released again at some point, i.e., the number of blocked tasks will be zero again eventually. Note that for these two queries we have added the condition $\neg(status = \texttt{S_HOME})$, which implies that the mission has not ended, yet. Otherwise no further messages are sent and receiving tasks remain in the waiting queue. To show that the *Main* task waits for messages from the input data queue, we have verified property (18), which states that whenever there is a receiving task (which in our case can only be the *Main* task) waiting to be served by the synchronous input data queue, the *Main* task is in its waiting state.

Properties (19) and (20) concern the asynchronous output data queue. To validate that a task waiting for data from the queue is eventually released, we have verified property (19). To validate that data is actually written into the output queue, we have verified that on all paths somewhere in the future the internal buffer of the data queue is not empty (20).

Properties (21) and (24) verify maximum times spent in the waiting state for all tasks, while using data queues.

Finally, to verify that our model can also cope with a high buffer pressure on data queues, we have increased the initial rate at which commands are generated in the model, and then successfully verified properties (25)–(26). Namely, that the buffer of the asynchronous output queue can become full and that if the buffer is full it will always eventually be not full again.

Table 2 gives an impression of the scalability of our approach by listing verification times and explored states for queries in Table 1. We compare two scenarios: the case where *bcet* equals *wcet* (as in [3]) and the case where *bcet* is less than *wcet*, as introduced in this paper. The last column shows the increase factors for time and explored states between the two cases.

The case study verified in this paper with $bcet < wcet$ demonstrates that reachability properties, i.e., the E□ properties (1)–(6) and (25), can be effectively verified for a given

real-time operating system modeled with our approach. Verification of these properties was achieved in mere milliseconds. The reason behind this rapid verification lies in the nature of E□ properties, which require the model checker to identify traces fulfilling the propositional formula often without necessitating an exhaustive search of the state space. This is also reflected in the small number of states explored required for these queries. A similar pattern is observed for the A◇ properties (12)–(15).

In contrast, the verification of properties using A□, namely (7)–(11), (16)–(19), (21)–(24), and (26), proved to be more demanding in terms of time and memory. Verification times varied significantly, ranging from 40 minutes to several hours. The most time-consuming query was (23), exceeding 7 hours, while query (19) involved the highest number of explored states, surpassing 127 million. These outcomes indicate that the state space for RTOS and applications modeled using our method can be substantial, especially for $bcet < wcet$. This is due to the inherent complexity of RTOS applications.

The comparison underscores that the scenario where *bcet* is less than *wcet* leads to a considerably larger state space, resulting in longer verification times and increased memory requirements, especially for the A□ properties. Here, the number of explored states increased with a factor between 12.05 (18) and 143.77 (26). Similarly, the verification times rose by factors ranging from 7.7 (8) to 1048 (26). Despite these steep increases in verification time and explored states, all properties were successfully verified for our model with a fixed mission and $bcet < wcet$. This demonstrates the applicability of our approach for formalization and verification of custom RTOS and their applications.

6 Related work

There exists a variety of sophisticated verification tools for software verification, for example, the CPAchecker [10], Frama-C [12], or VerCors [11]. These tools enable the automated or semiautomated formal verification of sequential or concurrent software. However, they abstract from the underlying RTOS as well as the timing behavior completely, yielding imprecise results if processes mainly interact via events or are heavily timing-dependant. Other approaches provide a complete formal RTOS model [21, 24], derive an RTOS from a given formal model [25], or compile RTOS source code into an RTOS model to check conformance of a real-time operating system according to specific standards [6]. However, transferring these approaches to existing RTOS implementations is challenging, as these typically lack formal semantics or a formal model.

There also exist several approaches to verify OSEK/VDX compliant systems. In [28], the authors present a TA model of a multitasking application running under a real-time operating system compliant with an OSEK/VDX standard. They have successfully verified timed and logical properties of the proposed model with the Uppaal model checker. In particular, they demonstrate that the timing analysis is more precise than a classical scheduling theory. However, they solely consider nonpreemptive scheduling, the model is not reusable for custom RTOS, and they do not provide reusable abstractions of general RTOS components. In [20], the authors present a CSP model of an OSEK/VDX RTOS kernel and verify various properties such as deadlock freedom. However, the application is not considered, and execution units with a higher precedence such as interrupt or cyclic handlers are disregarded. In [27], the authors present a formalization of the OSEK standard in Event-B and then verify RTOS implementations against the formalization. However, they again do not consider the application. In [13], the authors present an approach for the automatic verification of application-tailored OSEK kernels. Their key idea is to automatically compute an OS–application interaction graph from a given configuration and then to verify that it conforms to the standard. By generating the state transition graph statically, they avoid the state space explosion caused by thread interleaving. However, they disregard the concrete application and thus can neither verify properties of the application itself nor analyze its timing behavior. There exists a broad body of work for verifying schedulability using timed automata and extensions, e.g., [2, 16]. These approaches typically provide detailed models of scheduling strategies and tasks, e.g., for a specific RTOS implementation, but do not consider additional typical elements in RTOS, like events, communication or sensor information. Most closely related to our work are the approaches presented in [29–31] and [17, 18]. In [29–31], the authors construct an abstract model of an OSEK/VDX RTOS kernel, combine it with a translation of a given application, and then verify the resulting overall model with an SMT-based approach in [29] and with the SPIN model checker in [30, 31]. This work successfully demonstrates that real-time applications can be verified if the right abstractions are chosen. However, time and the inclusion of external information, like sensor data, is not considered. Most importantly, they also neither discuss possible generalizations nor the reusability of their formalization. In [17, 18], the authors use high-level Petri nets with stopwatches for modeling a multicore RTOS together with a real-time application. They analyze the schedulability of the resulting formal model using model checking. They successfully show the applicability of their approach using the open source implementation of the OSEK/VDX RTOS specification Trampoline [5]. While this approach is very well suited to investigate timing behavior, it does not consider event flags, sensor data, or communication via data queues.

7 Conclusion

In this paper, we have presented an extension of our previously proposed approach [3] to reduce the manual effort for the formalization of real-time applications that are developed with custom RTOS. Our key idea there was to provide reusable formal models and abstractions. We have presented TA templates that provide a configurable scheduler model, a generic task model that can control the execution of preemptable tasks, and a cyclic handler model that periodically executes nonpreemptable handlers. Other handlers (e.g., interrupt handlers) can analogously be modeled by replacing the periodic trigger with an external event. In addition, we provide reusable formalizations of typical interaction schemes, in particular the notification of events, waiting for events, and waiting for time. In this extended paper, we have additionally provided reusable and configurable timed automata models of synchronous and asynchronous data queues. For a given custom RTOS, the reusable models can be configured such that key RTOS components can be transformed into an abstract formal model. For a given real-time application, those can then be combined into a formal system model by transforming task and handler implementations with the help of the reusable formalizations of typical interaction schemes (e.g., for wait and sleep function calls, and for communication via data queues). The resulting overall model can be analyzed, formally verified, and graphically simulated using the UPPAAL tool suite.

To validate the applicability of our approach, we have configured our reusable formal models for the custom open source RTOS EV3RT and then translated a search & rescue robot implementation into a formal UPPAAL TA model. For now, this transformation was performed manually.

In a future work, we plan to use our approach also to validate the model with concrete input scenarios. In addition, we plan to validate the reusability of our proposed formal models with other custom RTOS. Furthermore, we plan to automate the transformation process by providing a transformation engine that should be configurable for various custom RTOS.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. TRON.μITRON4.0 specification (2007). https://www.tron.org/wp-content/themes/dp-magjam/pdf/specifications/en_US/TEF024-S001-04.03.00_en.pdf. Accessed: 2021-09-02
2. Abdeddaim, Y., Maler, O.: Preemptive job-shop scheduling using stopwatch automata. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 113–126. Springer, Berlin (2002)
3. Adelt, J., Gebker, J., Herber, P.: Towards reusable formal models for custom real-time operating systems. In: Formal Methods for Industrial Critical Systems: 27th International Conference, FMICS 2022, Warsaw, Poland, September 14–15, 2022, Proceedings. pp. 14–15. Springer, Berlin (2022)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
5. Bechennec, J.-L., Briday, M., Faucou, S., Trinquet, Y.: Trampoline an open source implementation of the OSEK/VDX RTOS specification. In: 2006 IEEE Conference on Emerging Technologies and Factory Automation, pp. 62–69. IEEE, New York (2006)
6. Béchennec, J.-L., Roux, O.H., Tigori, T.: Formal model-based conformance verification of an OSEK/VDX compliant RTOS. In: 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), pp. 628–634 (2018)
7. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL pp. 200–236. Springer, Berlin (2004)
8. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Lecture Notes on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Berlin (2004)
9. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal — a tool suite for automatic verification of real-time systems. In: Workshop on Verification and Control of Hybrid Systems. LNCS, vol. 1066, pp. 232–243. Springer, Berlin (1995)
10. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, pp. 184–190. Springer, Berlin (2011)
11. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods, pp. 102–110. Springer, Cham (2017)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: International Conference on Software Engineering and Formal Methods, pp. 233–247. Springer, Berlin (2012)
13. Deifel, H.-P., Göttlinger, M., Milius, S., Schröder, L., Dietrich, C., Lohmann, D.: Automatic verification of application-tailored OSEK kernels. In: Formal Methods in Computer Aided Design (FMCAD), Vienna, Austria. IEEE, New York (2017)
14. EV3RT Project. EV3RT (2019). https://ev3rt-git.github.io/about/
15. Gu, R., Shao, Z., Chen, H., Newman Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 653–669 (2016)
16. Han, P., Zhai, Z., Nielsen, B., Nyman, U.: Model-based optimization of ARINC-653 partition scheduling. Int. J. Softw. Tools Technol. Transf. **23**(5), 721–740 (2021)
17. Haur, I., Béchennec, J.-L., Roux, O.H.: Formal schedulability analysis based on multi-core RTOS model. In: 29th International Conference on Real-Time Networks and Systems, pp. 216–225 (2021)
18. Haur, I., Béchennec, J.-L., Roux, O.H.: Formal verification of the inter-core synchronization of a multi-core RTOS kernel. In: International Conference on Formal Engineering Methods, pp. 140–155. Springer, Berlin (2022)

19. Herber, P., Fellmuth, J., Glesner, S.: Model checking SystemC designs using timed automata. In: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS'08, Atlanta, GA, USA. pp. 131–136. ACM, New York (2008)
20. Huang, Y., Zhao, Y., Zhu, L., Li, Q., Zhu, H., Shi, J.: Modeling and verifying the code-level OSEK/VDX operating system with CSP. In: 2011 Fifth International Conference on Theoretical Aspects of Software Engineering, pp. 142–149. IEEE, New York (2011)
21. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: SeL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP'09, Montana, USA, pp. 207–220. ACM, New York (2009)
22. Laplante, P.A., et al.: Real-Time Systems Design and Analysis. Wiley, New York (2004)
23. OSEK. ISO 17356-3:2005 Road vehicles — open interface for embedded automotive applications — part 3: OSEK/VDX operating system (OS). International Organization for Standardization (2005)
24. Shi, J., He, J., Zhu, H., Fang, H., Huang, Y., Zhang, X.: ORIENTAIS: formal verified OSEK/VDX real-time operating system. In: 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems, pp. 293–301. IEEE, New York (2012)
25. Tigori, K.T.G., Béchennec, J.-L., Faucou, S., Roux, O.H.: Formal model-based synthesis of application-specific static RTOS. ACM Trans. Embed. Comput. Syst. **16**(4) (2017)
26. TOPPERS Project. Toyohashi open platform for embedded real-time systems. https://www.toppers.jp/en/project.html
27. Vu, D.H., Chiba, Y., Yatake, K., Aoki, T.: Verifying OSEK/VDX OS design using its formal specification. In: 2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 81–88. IEEE, New York (2016)
28. Waszniowski, L., Hanzálek, Z.: Formal verification of multitasking applications based on timed automata model. Real-Time Syst. **38**(1), 39–65 (2008)
29. Zhang, H., Aoki, T., Lin, H.-H., Zhang, M., Chiba, Y., Yatake, K.: SMT-based bounded model checking for OSEK/VDX applications. In: 2013 20th Asia-Pacific Software Engineering Conference (APSEC), vol. 1, pp. 307–314. IEEE, New York (2013)
30. Zhang, H., Aoki, T., Chiba, Y.: Verifying OSEK/VDX applications: a sequentialization-based model checking approach. IEICE Trans. Inf. Syst. **98**(10), 1765–1776 (2015)
31. Zhang, H., Li, G., Cheng, Z., Xue, J.: Verifying OSEK/VDX automotive applications: a spin-based model checking approach. Softw. Test. Verif. Reliab. **28**(3), e1662 (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.