Checking Consistency of Real-Time Requirements on Distributed Automotive Control Software Early in the Development Process using UPPAAL

Jan Toennemann[1], Andreas Rausch[1], Falk Howar[2], and Benjamin Cool[3]

[1] Clausthal University of Technology, Clausthal-Zellerfeld, Germany
[2] Dortmund University of Technology and Fraunhofer ISST, Dortmund, Germany
[3] Volkswagen AG, Wolfsburg, Germany

Abstract. The next generation of automotive control software will run on complex networks of control units, connected by a multitude of different bus systems. With a rising number of safety-critical functions being realized (at least partly) in software, real-time requirements for distributed functions become more important (e.g., time until a system reacts to a perceived driving situation). Defining and refining such requirements consistently during system development is not trivial. Inconsistencies or unrealizability can easily be introduced when decomposing requirements (e.g., time budgets) for functions that run on multiple control units. The automotive industry is actively pursuing methods for finding such problems as early as possible in the system design. In this paper, we present some initial work on the automated verification of requirements on distributed control functions that are deployed to networks of automotive control units. The presented analysis provides insights into the consistency of requirements and relies only on information available at the end of the planning stage in the development process.

1 Introduction

The next generation of automotive control software will run on complex networks of control units, connected by a multitude different bus systems. With a rising number of safety-critical functions being realized (at least partly) in software, real-time requirements for distributed functions become more important (e.g., time until a system reacts to a perceived driving situation). Established commercial analysis tools used to test automotive software systems, like TA Simulator [1] or SymTA/S [2], have recently added support for distributed functions. On the one hand, these tools are able to quite accurately simulate the system's behavior and often give very detailed results of tests in the form of exportable statistics and graphs. On the other hand, since the analysis is based on simulation, it requires a lot of information about the final system, e.g., statistical information about bus communcation delays obtained from actual recorded execution.

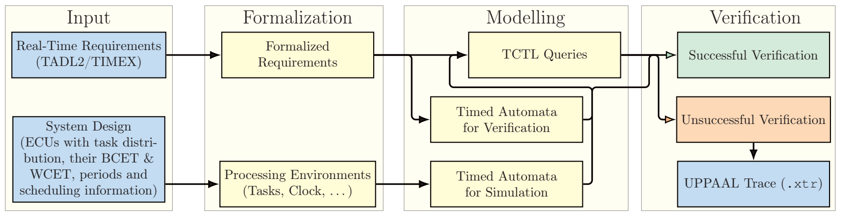
Input
Real-Time Requirements (TADL2/TIMEX)
System Design (ECUs with task distribution, their BCET & WCET, periods and scheduling information)
Formalization
Formalized Requirements
Processing Environments (Tasks, Clock, ...)
Modelling
TCTL Queries
Timed Automata for Verification
Timed Automata for Simulation
Verification
Successful Verification
Unsuccessful Verification
UPPAAL Trace (.xtr)

Fig. 1: Workflow for Checking Consistency of Timed Requirements with Information about System Design.

Moreover, each function or assistance system is mostly tested in isolation by its supplier as the behavior and timing influences of integrating a multitude of functions from various suppliers into a combined system is often very complex [17]. A reliable simulation of the actual behavior and checks on whether these requirements are fulfilled when integrating multiple functions on a network of control units can only be done very late in the development process [8].

Detecting inconsistencies at a stage this late in the development process can require major changes to the system as a whole and introduce a lot of additional work which may delay a project substantially, severely increasing the cost. As defining and refining real-time requirements consistently during system development is not trivial, the automotive industry is actively pursuing methods for finding such problems as early as possible in the system design. Inconsistencies or unrealizability can easily be introduced when decomposing requirements (e.g., time budgets) for functions that run on multiple control units. This calls for methods that enable analysis of real-time requirements even before a final system design is fixed, let alone implemented and ready to be tested.

While analysis in early stages of a project cannot be as precise as late in the process (during the planning phase, a non-negligible number of parameters affecting the final system is still unknown), an approach based on over-approximation of possible behavior can help to discover potential inconsistencies. Inconsistencies can then be addressed by refining requirements or assumptions on system behavior, e.g., when more precise timing information becomes available during the development process.

In this paper, we present such an approach for checking inconsistencies between multiple types of requirements early in the development process, requiring only little information about the final system. Figure 1 shows a high-level overview of the approach. We expect real-time requirements and basic information about the anticipated system design as input. We analyze the following types of requirements:

– Maximum execution time of tasks and reaction time of event chains,
– Data age of task output data,
– Periodicity of tasks,
– Scheduability, and
– Synchronization of tasks.

The analysis is based of the information listed below from the system design:

– A decomposition of functions into sub-functions assigned to control units,
– Data-flow between sub-functions,
– Assumed schedules for control units (based on OSEK or EDF), and
– Assumed best-case and worst-case execution times for sub-functions (including times for bus communication).

Requirements and system design are translated into a network of timed automata, encoding constraints on system behavior and TCTL queries representing proof obligations for requirements. The information about the automotive software system is then given as a system definition in UPPAAL [3] utilizing parametrizable templates, encoding scheduling constraints on possible executions and introducing clocks for observing some properties, as well as structures defined in the C-like language provided by UPPAAL. Consistency of requirements can then be evaluated through model checking. We demonstrate the approach on a (fictional) distributed brake-by-wire function.

Related Work. Timing constraints in automotive software systems, especially distributed systems, have been a field of extensive research in recent years and still continues to be. Analyzing and simulating the behavior of a single real-time system is not trivial, but has been reliably accomplished for single cores architectures. In recent years many manufacturers of control units have switched to a multi-core approach [12,18,10], where each processor has multiple cores and allows for parallel execution. It has been shown that it is still possible to completely simulate control units with multiple cores and parallel execution in order to ensure that the deployed software will perform reliably under all considered circumstances [5].

Considering a network of real-time systems introduces a whole new layer of complexity, resulting in a more complex simulation and analysis. For each new system introduced into the network an additional real-time clock needs to be considered, which might not run synchronous to that of the other systems in the network [19]. There exist various approaches to develop and test these interconnected systems and generally, the analysis of distributed real-time systems inside certain bounds can also lead to reliable results [7]. But the thorough analysis that is necessary for these results requires a large amount of data about the system, requiring both the system development as well as its implementation and configuration to be already finished when starting the tests.

Moreover, many tasks in automotive software systems do not run periodically in a fixed time grid, but are triggered by events in a non-deterministic matter [15]. Using classic model-checking, these cannot be reliably accounted for, since the worst case assumptions made in the process would be that the event is constantly triggered, resulting in an extremely overloaded system, which is not even close to situations that occur in real-world examinations. There are also propositions to only realize safety-critical functions using periodically triggered tasks [9,16], but as of right now, event-triggered tasks are considered to be an integral part of automotive software systems [15,17]. Several approaches exist to apply statistical analysis to include these type of tasks. The approach presented in [11] uses the experimental statistic model-checking toolkit integrated in the current UPPAAL

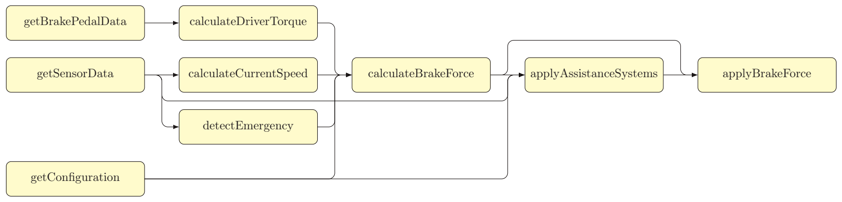

Fig. 2: Functional Decomposition of Brake-by-Wire Architecture.

development snapshots to verify such event properties inside UPPAAL. The model given in [11] is a very detailed representation of the system including a representation of the functional behavior in addition to the timing properties.

An approach similar to the one we present in this paper that does not rely on large amounts of data but rather on over-approximation of possible behavior has been used successfully used for modeling and verification of the CAN Bus in [14,6]. For later stages of a development process, when the bus design and message structure are fixed, the behavioral templates developed in these works could be incorporated into our approach in order to generate more realistic model of the underlying bus network.

Outline. The next section presents our motivating example, a distributed brake-by-wire function. Section 3 and Section 4 provide technical details on the phenomena that we model in templates and the types of properties that can be analyzed using these templates, respectively. Finally, we present results from an initial evaluation of the presented work on the motivating example in Section 5 before making some concluding remarks and discussing future work in Section 6.

2 Motivating Example

We will demonstrate the key ideas of our approach on the basis of a simplified brake-by-wire architecture (inspired by the example given in [4]). A brake-by-wire system replaces the mechanical connection between the driver and the vehicle by electronic systems [13]. These systems have a number of advantages over mechanical systems (e.g., weight reduction and increased cabin space) and are essential for autonomous driving. Replacing mechanical braking, the software in brake-by-wire systems falls into the highest safety integrity level with strong requirements imposed upon it, since any slightest error could endanger lives.

A high-level point of view of our simplified brake-by-wire architecture is shown in Figure 2. The distributed brake-by-wire function works by periodically polling the angle of the brake pedal to receive input from the driver, converting the angle to an amount of force that is applied to the brakes, applying additional assistance systems like electronic brakeforce distribution, and engaging the corresponding actuators in the brakes with the desired force. In addition to this driver-based brake routine, our system includes an emergency brake assistant, periodically analyzing data from various sensors of the car and activating the

Table 1: Mapping of Functions to Tasks for Brake-by-Wire Architecture including Estimated Time Budgets.

Task	**Function**	**BCET**	**WCET**	**Description**
τ_1	getBrakePedalData	3	4	Receive and store information about the current brake pedal angle
τ_2	getSensorData	6	7	Receive and store information about the sensors (accelero-, gyrometer, camera, ultrasonic sensor, . . .)
τ_3	getConfiguration	3	5	Receive and store information about the currently selected user options (engine recuperation, assistance systems, . . .)
τ_4	calculateDriverTorque	2	3	Calculate relative torque from brake pedal angle
τ_5	calculateCurrentSpeed	8	10	Use stored sensor data to calculate the current speed
τ_6	detectEmergency	16	22	Use stored sensor data to detect whether an emergency situation is imminent
τ_7	calculateBrakeForce	19	26	Combine current information from the brake pedal, sensors and settings to calculate the force to apply to the brakes
τ_8	applyAssistanceSystems	13	28	Apply enabled assistance systems based on currently stored sensor data and already calculated brake force
τ_9	applyBrakeForce	7	9	Apply the final result of the force calculation to the brakes by activating the brake actuators

brake actuators as fast as possible in case of an emergency. We assume several real-time requirements for the functions shown in the figure:

1. The function calculating the force that shall be applied to the brakes must always finish at most 28 ms after it started and the calculations must be done at least every 40 ms.
2. The sensor data used by the assistance systems may at most be 12 ms old and the already pre-processed data from the brake pedal may at most be 16 ms old when the calculations of the brake force start.
3. The driver-triggered brake routine, from the polling of the brake pedal angle to the finished activation of the brake actuators, must finish within 110 ms.
4. The path from the main brake controller calculating the brake force up to the finished activation of the actuators may at most take 80 ms and the input data to the function calculating the brake force must always be from within a time frame of 10 ms.
5. The emergency brake routine, from the polling of the sensor data up to the finished activation of brake actuators, may never surpass a total of 85 ms.

Since we are dealing with automotive software systems, we consider a network of control units using a real-time operating system, where functions are implemented using periodically triggered tasks with deterministic scheduling; we will call such a software system a *processing environment*. We assume that we have a homogeneous hardware architecture, where all tasks have a fixed best-case and worst-case execution time (BCET & WCET, respectively) regardless of the processing environment they are currently deployed on. Due to networking and wiring constraints, we assume each additional processing environment introduced into the system to have a slight clock offset of 1 ms compared to the one added before, such that the offset between two processing environments pe_n, pe_m can be calculated as $m - n$ ms, making the processing environment with the lowest index the reference system.

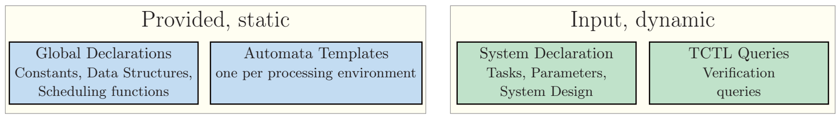

Fig. 3: Resulting Documents of UPPAAL Modeling.

Sub-functions are described in Table 1. The noted BCET and WCET are over-approximated and would – in case of a consistent system – be handed to suppliers as timing requirements for the developed functions, since they are then a fundamental base for a consistent system. Possible inconsistencies range from rather simple cases, for example that the time grid assigned to a task fails to make the function run as often as needed for a periodicity requirement to be met, to very complex cases, like for instance when a group of tasks that needs to be run sequentially in a given amount of time (an event chain) does not finish fast enough in edge cases, e.g. when various offsets correlate in a way that is not instantly obvious as the worst-case.

3 Modeling

In this section, we walk through the development process of the templates which we use to simulate the timing behavior of distributed automotive software systems. In order to use UPPAAL as a model-checking tool, we require a model of automotive software systems that can be used as input for the verification. Using UPPAAL global declarations, we have created basic data structures like tasks and implemented functions for handling these as well as the scheduling. The processing environments are represented using templates, a combination of a modeled timed automaton and accompanying code in the C-like language provided by UPPAAL; an overview of the created documents is shown in Figure 3. Before setting up the automata and their behavior, we will use the global declarations to implement a task model and scheduling functions, which can then be accessed from the templates.[4]

Tasks. To be able to refer to tasks, we use the basic structure `Task` composed of a numeric identifier, the `ID` of the task, and the `BCET` and `WCET` as well. Since we require tasks to have scheduling information, the structures `EDF_Task`, comprised of a task, a relative deadline and a period, as well as `OSEK_Task`, consisting of a task, priority and period tuple, were created.

Tasks are not spawned directly but rather as instances, defined as `EDF_Task_Instance` and `OSEK_Task_Instance`, both of which allow us to save their execution time as well as their start time; additionally, the `EDF_Task_Instance` also saves the absolute deadline, which is the absolute deadline of

[4] The complete UPPAAL models can be accessed at `http://www2.in.tu-clausthal.de/~jtoennemann/uppaal-2018-01/`, here we will only cover the most relevant parts to understand the verification process.

```
int[0, TASK_QUEUE_MAX] EDF_schedule(EDF_Task_Queue &tq) {
  EDF_Task_Instance next_eti = tq[0];
  int[0, TASK_QUEUE_MAX] next_eti_pos = 0;
  int[1, TASK_QUEUE_MAX + 1] i = 1;
  while(i < TASK_QUEUE_MAX && tq[i] != NULL_EDF_TI) {
    if(tq[i].deadline < next_eti.deadline) {
      next_eti = tq[i];
      next_eti_pos = i;
    }
    i++;
  }
  return next_eti_pos;
}
```

Listing 1.1: UPPAAL Code for EDF Scheduling.

the instance calculated from the start time and the relative deadline of the corresponding `EDF_Task`.

For each task we simulate, we want information about its runtime (its execution time, the time that has passed since the start of the execution) as well as its data age (the time that has passed since the task last finished executing and thus provided new output data). Since we assume tasks to be unique (in the sense that each task ID is only assigned once, globally), we store the runtime and data age clocks using arrays in the global declarations.

We also add broadcast channels to notify of the beginning and end of task execution as well as a Boolean array to save whether a task is currently being executed, as we are unable to compare clock values outside of guards in UPPAAL, including custom functions.

Scheduling. To create an EDF task instance, we only pass the corresponding task as a reference and the local time of the processing environment as the parameters, since the rest can be calculated from there. The absolute deadline is calculated from the local time of the task-spawning PE and the relative deadline of the EDF task, the start time can be assumed to be the passed local time, and the execution time is zero, since the instance has just been newly generated.

With an initialized system, we are working with EDF task instances in the task queue represented by the data type `EDF_Task_Instance`. The queue is represented as an array, initialized with the elements `NULL_EDF_TI`, ordered in a way that we can consider the first encountered `NULL_EDF_TI` to be the end of the queue. This is a consistency requirement needed by the functions used to enqueue and dequeue the task instances.

Since each task instance in the queue has information about its absolute deadline, the scheduling function shown in Listing 1.1 simply moves through the queue and returns the instance with the lowest absolute deadline, that is, the instance that needs to be finished next. Should multiple instances have the same deadline, the index of the first one encountered is returned, which is the one with a lower index. The scheduling function does not return the instance, but rather its position in the task queue of the processing environment. This is due to a limitation in UPPAAL, which – while allowing references to be passed to a

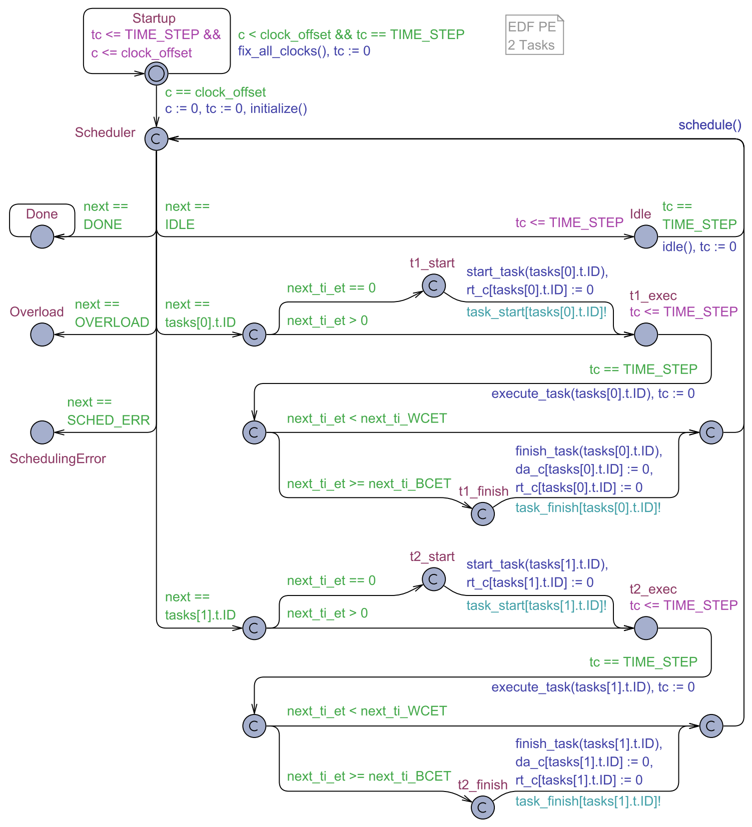
Startup
tc <= TIME_STEP &&
c <= clock_offset
c < clock_offset && tc == TIME_STEP
fix_all_clocks(), tc := 0
EDF PE
2 Tasks
c == clock_offset
c := 0, tc := 0, initialize()
Scheduler
schedule()
Done
next ==
DONE
next ==
IDLE
tc <= TIME_STEP
Idle
tc ==
TIME_STEP
idle(), tc := 0
t1_start
start_task(tasks[0].t.ID),
rt_c[tasks[0].t.ID] := 0
task_start[tasks[0].t.ID]!
next_ti_et == 0
next_ti_et > 0
t1_exec
tc <= TIME_STEP
Overload
next ==
OVERLOAD
next ==
tasks[0].t.ID
tc == TIME_STEP
execute_task(tasks[0].t.ID), tc := 0
next ==
SCHED_ERR
SchedulingError
next_ti_et < next_ti_WCET
finish_task(tasks[0].t.ID),
da_c[tasks[0].t.ID] := 0,
rt_c[tasks[0].t.ID] := 0
task_finish[tasks[0].t.ID]!
next_ti_et >= next_ti_BCET
t1_finish
t2_start
start_task(tasks[1].t.ID),
rt_c[tasks[1].t.ID] := 0
task_start[tasks[1].t.ID]!
next_ti_et == 0
next_ti_et > 0
t2_exec
tc <= TIME_STEP
next ==
tasks[1].t.ID
tc == TIME_STEP
execute_task(tasks[1].t.ID), tc := 0
next_ti_et < next_ti_WCET
finish_task(tasks[1].t.ID),
da_c[tasks[1].t.ID] := 0,
rt_c[tasks[1].t.ID] := 0
task_finish[tasks[1].t.ID]!
next_ti_et >= next_ti_BCET
t2_finish

Fig. 4: EDF-scheduled Processing Environment Template with two Tasks.

function – does not allow a function to return a reference. A way to circumvent this would be to pass another reference and set this to the selected instance, but both for consistency and compatibility reasons we chose the approach of just returning the index in the queue. With OSEK instances, scheduling works similar to the presented EDF function, using priority as a parameter instead of a relative deadline.

Templates. We have developed parametrizable templates to represent processing environments, such that we only need to create one such template per amount of tasks run on the system and insert the tasks dynamically using the parameters. The templates itself do not differ based on the scheduling function used, only in the function declarations. All templates share common definitions, two clocks independent from the number of tasks handled (one for the automaton's local time, one for counting to the next time step during any task's execution), a constant for the amount of tasks handled by the template, a variable saving the current local time and an array of the contained task's periods, used as triggers for adding them to the queue. In addition to these, shorthands are defined to access the next action, determined by the scheduler, for the queue index of the next task instance, the amount of task instances currently in the queue and the execution time of the next task instance, along with the corresponding tasks' WCET and BCET.

Not only definitions, but also some functions are common to all templates. Several functions are used to reset clocks after each tick, e.g. to only allow runtime clocks to progress during actual task execution. While these functions are present in all templates, regardless of the scheduling algorithm used, there are minor differences in the implementations because of different data types. Different functions are run for each task start, finish and execution step.

The generic template for an EDF-scheduled processing environment running two tasks shown in Figure 4 calls the functions `initialize()` upon system initialization, `schedule()` after each time step and `idle()` when leaving the *Idle* location. For each task, there is a compound of six locations supplied with the numerical task identifiers; these call the functions `start_task(ID)`, `execute_task(ID)` and `finish_task(ID)` and are also responsible for sending over the relevant broadcast channels for the task ID.

It is worth noting that the automaton uses additional auxiliary locations, namely the *Done*, the *Overload* and the *SchedulingError* locations. The first two are the result of limitations in UPPAAL, as both clock values and integer variables are limited by the `int16` bounds of the underlying C architecture, which amounts to 32767. This means that neither a clock value, nor the value of a local variable can rise above this bound. Since each automaton has its own clock, continuously counting the local time, this bound effectively limits the simulation steps possible. We have found a bound simulation time sufficient for the systems we tested and since a limit on the time reduces the state space, setting it to even lower values than the allowed maximum also allows for faster verification. This is why the automata network, after having reached the time set by the `TIME_MAX` constant, forces all automata to enter the *Done* location, effectively reaching a verifiable end condition.

As long as all possible variations occur at least once during the time specified by `TIME_MAX`, this is not an issue. For systems with very large, different periods, issues might be encountered as not all combinations might be part of the simulation and verification process. Since this is a problem with the underlying architecture, it cannot be fixed trivially. A possible mitigation might be to introduce an additional integer variable into the system and increment it at a fixed interval, for example each 30000 time steps, resetting the clock in the process. This allows to keep track of way larger time spans, but also requires several changes to the model.

System Declarations. The system declarations are the last set of declarations in UPPAAL and are used to describe the system, which is the network of timed automata that shall be simulated and verified. For this system, the processing environments need to be defined by instantiating the relevant templates. The instantiation of templates works similar to the creation of objects

```
// Task definitions (ID, BCET, WCET)
const Task T1 = {1, 3, 4};   // getBrakePedalData
const Task T2 = {2, 6, 7};   // getSensorData
...
const OSEK_Task OT3 = { T3, 1, 30 }; // OSEK Task Definitions
const OSEK_Task OT4 = { T4, 2, 30 }; // (Task, Priority, Period)
const OSEK_Task OT6 = { T6, 3, 30 };
const OSEK_Task OT7 = { T7, 1, 30 };
const OSEK_Task OT8 = { T8, 1, 30 };
const EDF_Task ET1 = { T1, 15, 30 }; // EDF Task Definitions
const EDF_Task ET2 = { T2, 20, 30 }; // (Task, rel. Deadline, Period)
const EDF_Task ET5 = { T5, 25, 30 };
const EDF_Task ET9 = { T9, 30, 30 };
const EDF_Task PE1_Tasks[4] = { ET1, ET2, ET5, ET9 }; // Array
const OSEK_Task PE2_Tasks[3] = { OT3, OT4, OT6 };     // Compositions
const OSEK_Task PE3_Tasks[1] = { OT8 };
const OSEK_Task PE4_Tasks[1] = { OT7 };
PE1 = PE_4T_EDF(PE1_Tasks, 0);  // PE Definitions Template
PE2 = PE_3T_OSEK(PE2_Tasks, 1); // (Task Array, Offset to Reference PE)
PE3 = PE_1T_OSEK(PE3_Tasks, 2);
PE4 = PE_1T_OSEK(PE4_Tasks, 3);
```

Listing 1.2: System Declaration for some Tasks from the Motivating Example.

in object-oriented languages, the parameters are given during creation. After all templates have been instantiated, the simulatable system must be defined using the `system` directive and a comma-separated list of already created templates. An excerpt of the system definitions covering the tasks of the motivating example is given in listing 1.2. These structures cover just the basic task information; for schedulability, they need to be embedded in either EDF or OSEK tasks and passed to processing environments in arrays.

4 Verification

In this section, we describe how safety properties can be encoded as TCTL queries using the the small example system from Section 3 for illustration. The verification of synchronicity (i.e., multiple tasks finishing within a given period of time) was realized but is not covered here in detail due to space constraints. Due to the way the clocks in timed automata work, each task runtime clock has a valuation of $v(c) \in [0, 1]$ when the corresponding task's simulation state is currently neither executing nor suspended. As a consequence, we are unable to reliably check whether a task has actually just started execution based on the runtime clocks and need to resort to the location names and the broadcast channels. Since UPPAAL itself does not allow dynamic or parametrizable location names, this needs to be taken into account for several verification queries. The location names used are from the task compounds shown in Figure 4 (e.g., `t1_start`) and the indices that need to be used are given using italicized mathematic notation. Requirements are specified using function notation, where each requirement is represented using a function over one or multiple tasks.

4.1 Verification of Properties Using TCTL

We start by covering real-time requirements which we can verify using TCTL queries and the simulated network of timed automata representing processing environments. The requirements covered here are requirements over a single or over two tasks.

Maximum Execution Time of a Task. We will consider the *maximum execution time* of a task to describe the maximum amount of time that is allowed to pass between the start and finish events of any pair of the task's instances. Due to the existence of runtime clocks, the maximum execution time can easily be verified. As the UPPAAL model is time-bound by the constant `TIME_MAX`, we need to prepend a condition to account for this upper bound. Otherwise a system state in which the requirement is not fulfilled can always be found outside of the valid time bounds, as the automata enter the *Done* state and do not continue resetting the clocks.

We have several clocks to choose from that can act as global clocks to compare to this time bound, mainly the clock of the reference system and the runtime or data age clocks of task ID 0; since PE IDs start at 1, those start at the beginning of the simulation and are never reset. Since the reference system may be declared with varying names, we will use `rt_c[0]` as the global clock for the following verification queries.

Assuming a specific task is represented using a task with the ID n in the UPPAAL model, we use the query

`A[] (rt_c[0] <= TIME_MAX) imply (rt_c[`n`] <= `$\mathrm{MET}(\tau_n)$`)`

to check for validity of the requirement $\mathrm{MET}(\tau_n)$.

Maximum Data Age. We define the *maximum data age* to express the maximum amount of time that may pass between the finish event of one task τ_n and the start event of another task τ_m, essentially the age of the output data provided by τ_n used as input by τ_m. Just like with the execution time, the verification of the data age requirement was made easy in the model-building process by introducing the relevant clocks. Considering two tasks with IDs n, m deployed on the same processing environment with ID i, we can apply the template

`A[] (PE`i`.t`j`_start imply (da_c[`n`] <= `$\mathrm{MDA}(\tau_n, \tau_m)$`))`

to check whether $\mathrm{MDA}(\tau_n, \tau_m)$ is upheld by the given system; where j is the index of task m on the processing environment.

Periodicity We assume the periodicity requirement to be describing the maximum amount of time that may pass between two finish events of the same task. The verification of this requirement can be achieved easily as well, due to the fact that the data age clock is reset in the time step *after* the finish state, not before. Using the UPPAAL query

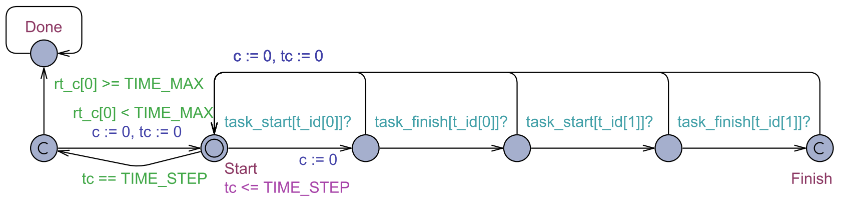

Fig. 5: Template for Verification of an Event Chain of Two Tasks.

`A[] (PE`i`.t`j`_finish imply (da_c[`n`] <= (`$\mathrm{PER}(\tau_n)$`))))`

we can verify whether a task with ID n running on the processing environment with ID i and the task array index j satisfies the requirement $\mathrm{PER}(\tau_n)$.

Schedulability and Queue Overload. While not specifically a requirement, we can check whether a system might encounter an error during the simulation, namely a runtime scheduling error or a queue overload. All auxiliary error states in the templates are states with no outgoing edges. While the *Done* state of a processing environment template can transition to itself, this is not possible in the error states. Since these error states are the only states in the whole automaton without outgoing edges, we can check whether it is possible for the automaton to encounter a deadlock. To check whether a network of automata proceeds through the whole bound simulation without the possibility of any automaton entering an error state, we use the query "`A[] not deadlock`" to check for deadlock-freeness.

Task Execution. If there is a state in which the runtime clock of a task has a valuation exceeding that of a single time step, we know that the task was executed at least once as there exists at least one system state in which the corresponding runtime clock was not reset; note that this only works if the WCET of the task is indeed larger than the duration of a single tick. To check whether a task instance of a task with ID n is actually ever executed, we can use the query

`E[] (rt_c[0] <= TIME_MAX) and (rt_c[`n`] > 1)`

using a conditional prefix depending on the time limit.

4.2 Verification Using Additional Automata and TCTL

Properties that require information about a time span rather than a single point in time or need to react based on previous input or actions that cannot be expressed using simple TCTL queries. All properties detailed in Section 4.1 expressed that either at a certain point in time or at all times the corresponding property must hold. For these requirements, we will introduce additional automata into the simulated system to have the ability to react to multiple events in a single verification run, enabling state-aware verification for our model, which is necessary to verify both the synchronization constraint as well as the maximum reaction time requirement. Note that both of these can range over an arbitrary amount of tasks and span multiple events over a time span.

Maximum Reaction Time of an Event Chain. We consider event chains to be a sequentially ordered set of events and only consider task start and finish events here, such that we can also create parametrizable templates for event chains. The idea is to move through the locations by reacting to the start and the finish events of the contained tasks. To actually catch all valid flows through the event chain, we introduce non-determinism, such that the event chain automaton can switch to the start location from every other location using a non-guarded transition, resetting its internal clock as well as its tick clock. As the automaton is required to transition when receiving on the broadcast channel, and because the non-determinism introduces the ability for the automaton to always return to its start location, verification of the safety property using `A[]` are used to ensure that there is no single valid event chain flow that violates the requirement.

As parameters to the template, we can simply pass an array of numeric task IDs representing the tasks in the order they appear in their event chain. For an event chain template of `n` tasks, we define the parameters as `const int[1, TASK_AMOUNT] t_id[n]`, for `n` = 2 the template would look like Figure 5. The code of each event chain automaton template simply contains `clock c; clock tc;` and does not require any adjustments when changing the amount of tasks. When the automaton is properly defined, we can check for the MRT requirement using the event chain automatons internal clock. Given an event chain ec_n, we can use a query

`A[] (ecn.Finish imply ecn.c <=` $\mathrm{MRT}(ec_n)$`)`

to check whether each complete run of the event chain was within the specified bounds. This of course only works when the event chain does reach a finish state, which might not be true in every case. A query to check whether this happens is `E<> ecn.Finish` (because of the non-determinism `E<>` needs to be used instead of `E[]`).

5 Evaluation

In this section, we briefly demonstrate how the presented approach can be applied using our motivational example. We have formalized the textual requirements from Section 2 and are using the system design from Listing 1.2, specified verification automata and TCTL queries. To limit the size of the reachable state space, we we only use the automata necessary for the verification of each property during model checking.

Using this, we can check all established requirements. Table 2 shows an overview of the properties, the queries and their results. As indicated, most requirements are not met in the initial design of the brake-by-wire system. In these cases, using UPPAALs *Diagnostic Trace* option, we can get a snapshot of

Table 2: TCTL Queries and Verdicts for the Properties from Section 2.

Property	UPPAAL Query	Verdict
$\mathrm{MET}(\tau_7) = 28$	`A[] (...) imply (rt_c[7] <= 28)`	✓
$\mathrm{PER}(\tau_7) = 40$	`A[] PE4.t1_finish imply (da_c[7] <= 40)`	✓
$\mathrm{MDA}(\tau_2, \tau_8) = 12$	`A[] PE3.t1_start imply (da_c[2] <= 12)`	✗
$\mathrm{MDA}(\tau_4, \tau_7) = 16$	`A[] PE4.t1_start imply (da_c[4] <= 16)`	✓
$\mathrm{MRT}(ec_1) = 110$	`A[] ec1.Finish imply ec1.c <= 110`	✗
$\mathrm{MRT}(ec_2) = 85$	`A[] ec2.Finish imply ec2.c <= 85`	✗
$\mathrm{MRT}(ec_3) = 80$	`A[] ec3.Finish imply ec3.c <= 80`	✗
$\mathrm{SYNC}(\tau_3, \tau_4, \tau_5) = 10$	`A[] not sync1.Error`	✗

the automata network in a state where the requirement is violated. This helps us with identifying the root cause of the inconsistencies, assisting in the development of a system consistent with all requirements. More details and refined system designs that satisfy more preoperties are available online.[5]

We were able to analyze all presented properties of the brake-by-wire example. However, already when analyzing properties on this small example, it became obvious that the performance of model checking approach will need to be optimized heavily in order to apply our approach to large-scale industrial processes and systems with several hundred functions and tens of control units. **PSPACE**-completeness of model-checking using Timed Automata and TCTL can be partly mitigated by only incorporating the automata required for the verification into the system for each query, but when complex requirements like the reaction time of an event chain need to be checked on very large systems, either another modelling approach or a additional assumptions that reduce the state space may become necessary.

6 Conclusion

We have identified a set of data that allows to model-check the consistency of real-time requirements in distributed software systems using UPPAAL early in the development process — and especially long before precise simulations are feasible. With the provided set of UPPAAL templates, multiple timing requirements over such a system can be checked for inconsistencies and used as an indicator whether the basic assumptions require any modification. The proposed approach can be extended by adding clocks to capture the time between occurrences of other events, as well as the implementation of supplementary task properties, the model can be extended to incorporate a more timing-related behavior and allow for the verification of several other requirements, like the maximum core execution time of a task, which describes the execution time while ignoring the time the task spends in a suspended state. We have demonstrated the approach on a small brake-by-wire system. In a next step, we plan to evaluate performance and scalability in actual distributed automotive software systems.

[5] `http://www2.in.tu-clausthal.de/~jtoennemann/uppaal-2018-01/`

References

1. SymTA/S. `https://auto.luxoft.com/uth/timing-analysis-tools/`.
2. TA Simulator. `https://www.timing-architects.com/en/solutions/ta-tool-suite/ta-simulator/`.
3. UPPAAL. `http://www.uppaal.org/`.
4. Hans Blom, Dr. Lei Feng, Dr. Henrik Lnn, Dr Johan Nordlander, Stefan Kuntz, Dr. Bjrn Lisper, Dr. Sophie Quinton, Dr. Matthias Hanke, Dr. Marie Agns Peraldi-Frati, Dr. Arda Goknil, Dr. Julien Deantoni, Gilles Bertrand Defo, Kay Klobedanz, Mesut zhan, and Olha Honcharova. Timing model tools, algorithms, languages, methodology, use cases. Technical report, 2012.
5. Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer US, 2011.
6. Beomyeon Cho, , Taewook Kim, and Jin-Young Choi. CAN database verification framework using UPPAAL. *International Journal of Computer Theory and Engineering*, 9(6):438–442, 2017.
7. T. Cucinotta, A. Mancina, G.F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina. A real-time service-oriented architecture for industrial automation. *IEEE Transactions on Industrial Informatics*, 5(3):267–277, aug 2009.
8. Patrick Frey. *A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems*. PhD thesis, 2011.
9. Jelena Frtunikj. Safety framework and platform for functions of future automotive e/e systems. *Automotive and Engine Technology*, jul 2016.
10. Thomas Fuhrman, Shige Wang, Marek Jersak, and Kai Richter. On designing software architectures for next-generation multi-core ecus. *SAE Int. J. Passeng. Cars Electron. Electr. Syst.*, 8:115–123, 04 2015.
11. Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikučionis, and Petur Olsen. Formal analysis and testing of real-time automotive systems using UPPAAL tools. In *Formal Methods for Industrial Critical Systems*, pages 47–61. Springer International Publishing, 2015.
12. Patrick Leteinturier, Simon Brewerton, and Klaus Scheibert. Multicore benefits & challenges for automotive applications. In *SAE Technical Paper*. SAE International, 04 2008.
13. Chris Line, Chris Manzie, and Malcolm Good. Control of an electromechanical brake for automotive brake-by-wire systems with an adapted motion control architecture. In *SAE Technical Paper Series*. SAE International, may 2004.
14. Can Pan, Jian Guo, Longfei Zhu, Jianqi Shi, Huibiao Zhu, and Xinyun Zhou. Modeling and verification of CAN bus with application layer using UPPAAL. *Electronic Notes in Theoretical Computer Science*, 309:31–49, dec 2014.
15. Achim Rettberg, Mauro C. Zanella, Michael Amann, Michael Keckeisen, and Franz J. Rammig, editors. *Analysis, Architectures and Modelling of Embedded Systems*. Springer Berlin Heidelberg, 2009.
16. Florian Sagstetter. *Schedule Synthesis for Time-Triggered Automotive Architectures*. Dissertation, Technische Universitt Mnchen, Mnchen, 2016.
17. Oliver Scheickl. *Timing Constraints in Distributed Development of Automotive Real-time Systems*. Dissertation, Technische Universitt Mnchen, Mnchen, 2011.
18. Rolf Schneider, Simon Brewerton, and Denis Eberhard. Multicore vs safety. In *SAE Technical Paper*. SAE International, 04 2010.
19. H. Thane and H. Hansson. Testing distributed real-time systems. *Microprocessors and Microsystems*, 24(9):463–478, feb 2001.