Journal of Universal Computer Science, vol. 20, no. 14 (2014), 1903-1925
submitted: 20/3/14, accepted: 28/11/14, appeared: 1/12/14 © *J.UCS*

Formal Modeling and Verification of Motor Drive Software for Networked Motion Control Systems

Youngdong Kim[1]
(University of Seoul, Seoul, Korea
ydkim@uos.ac.kr)

Ikhwan Kim
(University of Seoul, Seoul, Korea
ihkim@uos.ac.kr)

Inhye Kang
(University of Seoul, Seoul, Korea
inhye@uos.ac.kr)

Taehyoun Kim
(University of Seoul, Seoul, Korea
thkim@uos.ac.kr)

Minyoung Sung
(University of Seoul, Seoul, Korea
mysung@uos.ac.kr)

Abstract: This paper presents a model-based approach to the design and verification of motor drive software for networked motion control systems. We develop a formal model for an Ethernet-based motion system, where, using timed automata, we describe the concurrent and synchronized behaviors of the components, i.e., motion controller, motor drives, and communication links. The drive, in particular, is modeled in enough detail to accurately reflect the software implementation used in a real drive. We use the design of multitasked drive software with fixed-priority preemptive scheduling. With UPPAAL model checking, we verify the precision and accuracy of the rendered motion in terms of the requirements on the actuation delay at each drive and the actuation deviation between different drives, respectively. The analysis results demonstrate the benefits of our model-based approach in the safety verification and design space exploration of motor drive software. We show that it is possible to verify deadlock freeness and real-time schedulability in an early design phase. And, for varying number of drives and size of messages, we can successfully determine the combination of task periods that leads to the best precision and accuracy.

Key Words: Timed automata, Formal methods, Motor drive software, Formal methods, Actuation delay and deviation.
Category: D.2.4, J.7

[1] He is currently with Hyundai Autoever Corp., Seoul, Korea (ydkim@hyundai-autoever.com).

1 Introduction

Being used in various mechanical applications such as industrial automation, robotic surgery, vehicles, and military equipment, motion control systems are becoming increasingly important. A typical motion system comprises a motion controller and motor drives that cooperate closely with each other. Since the motion controller should be able to control every motor precisely and synchronize its operation accurately with others, it has been preferred to connect each motor to the controller one-to-one using dedicated links [Yu et al. 2009]. Recently, however, as the high-speed industrial Ethernet is rapidly adopted by the industry, Ethernet-based motion systems are gaining ground [Kim et al. 2012, Vitturi et al. 2011, Benzi et al. 2005].

Networked motion systems have stringent timing constraints. Two of the fundamental ones are the requirements on the end-to-end actuation delay and deviation. The actuation delay is defined as the time taken from the dispatch of a command at the controller to the corresponding actuation by a motor drive. The delay affects the minimum cycle time at the controller. Since a shorter cycle time usually contributes to the precision in controlling individual drive, it can be said that, the shorter the actuation delay is, the higher becomes the *precision* of single-axis motion. The actuation deviation is defined as the time difference between the earliest and latest actuation at different drives in response to the commands of the same controller cycle. Similarly, it can be said that, the smaller the actuation deviation is, the higher becomes the *accuracy* of multi-axis motion.

With real-time Ethernet that provides high-speed deterministic delivery of control messages, the end-to-end delay primarily relies on the time taken by the drive to actuate in response to the controller command. The growing complexity of modern motion systems demands sophisticated control software with concurrent tasks. In such systems, the constraints from desired motion directly relate to the design of the multitasked drive software. Thus, in order to achieve the highest possible precision and accuracy, a systematic approach is required for the task decomposition and period synthesis, whose derived design should maximize the utilization of the drive hardware while satisfying the functional and timing requirements.

Formal methods provide the mathematical and logical foundations for the specification and verification of software design. Formal methods have been used for decades in various applications including nuclear engineering, medical systems, transport, and military systems [Woodcock et al. 2009, Pajic et al. 2012, Lahtinen et al. 2012, Choi 2013, Bon and Dutilleul 2013, Min et al. 2013]. Although there exist other relevant methods such as scenario-based tests and simulation, formal modeling and verification is recommended because it better copes with the ever-increasing system complexity and scale. *Timed automata*, in particular, is used for the formal specification and verification of real-time systems. Timed automata has been applied for many mission-critical applications, successfully verifying the safety of, for instance, medical systems [Pajic et al. 2012] and nuclear control systems [Lahtinen et al. 2012].

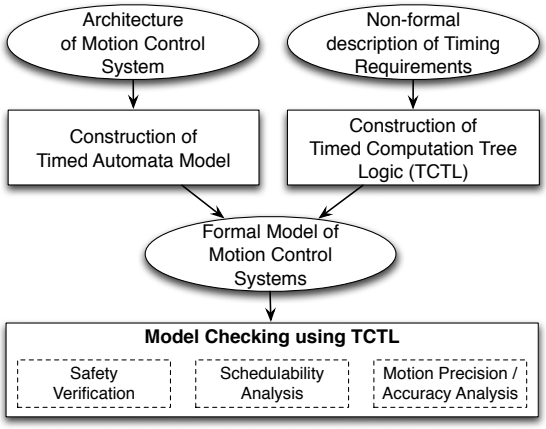


Figure 1: Our model-based approach.

This paper introduces a model-based approach for the design of an Ethernet-based motor drive. Using timed automata, we describe the timing behavior of the motion control system, and formulate the precision and accuracy of the rendered motion in terms of the timing requirements on the end-to-end actuation delay and deviation. The drive, in particular, is modeled in enough detail to accurately reflect the software implementation that is used in a real drive. For this purpose, we use the design of multitasked drive software with fixed-priority preemptive scheduling [Kim et al. 2012]. The developed automata model is then effectively utilized in the design space exploration of the motor drive software for the synthesis of task periods. The analysis results using UPPAAL tool [UPPAAL] show that it is possible to verify the deadlock freeness as well as the schedulability of real-time tasks. And, for varying number of drives and size of messages, we can successfully determine the combination of task periods that leads to the best precision and accuracy. Figure 1 succinctly illustrates our model-based approach.

To the best of our knowledge, our work is the first to formally model and verify the motion control system using timed automata. It offers numerous benefits summarized as follows:

- It is possible to assure the correctness of control system specifications in early stages of the design. In our case study, the timed automata model facilitated the verification of deadlock-freeness and real-time schedulability of motor drive software.
- The model-based approach supports efficient design space exploration. Our analysis results show that, for varying number of drives and size of messages, we could successfully determine the combination of best task periods together with the predicted values of actuation delay and deviation.
- By simulation-based tracing and model checking, we can easily produce and correct potential design problems.

The remainder of this paper is organized as follows. Section 2 reviews the back-

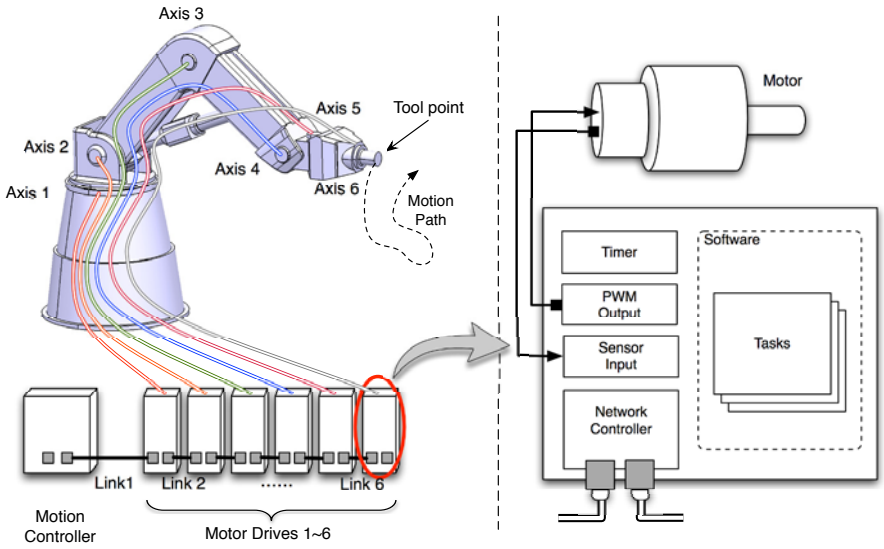


Figure 2: Industrial robot, an example of Ethernet-based motion systems.

ground to networked motion systems and timed automata and presents the related works. In Section 3, we describe our timed automata model for the motion control system, and in Section 4, we verify the timing requirements and discuss the drive software design. And finally, Section 5 concludes the paper.

2 Background

2.1 Ethernet-based Motion Control System

The networked motion system considered in this paper is composed of a controller and a number of homogeneous motor drives which are interconnected with each other using industrial Ethernet. Figure 2 shows an example of such a motion system, an industrial robot with six degrees of freedom. The motion controller periodically transmits to the motor drives real-time messages with the commands of target position or velocity. Then, based on the command information, each drive operates its control loop and actuates the associated axis by generating proper PWM (Pulse-Width Modulation) signal on the motor. The drive is also responsible for the feedback of motor status. It reads the attached sensor inputs and sends back the information to the controller via the real-time messages. The status information includes current position and velocity, which are used by the controller for the computation of commands in the next cycle. By coordinating the multi-axis actuation, the motion system can make the tool point follow the desired trajectory.

For the design of the software that is used in the motor drive, we have two challenges to cope with. First, since a motor drive is typically used for mission-critical systems, it

is essential to verify its safety and timing correctness. For this purpose, we should be able to ensure the deadlock freeness as well as the schedulability of real-time tasks of the software design. In particular, we must guarantee that the tasks for motor actuation and real-time communication do not miss their deadlines. Second, it is required to be able to predict and minimize the time for the response to the controller command. To achieve a high precision in a single-axis control, the motor drive should be as responsive as possible to the command. At the same time, to achieve a high accuracy in the coordinated motion, the responses by different drives should be highly synchronized. In this paper, we relate the precision and accuracy of the rendered motion to the timing requirements on the end-to-end actuation delay and deviation. We define the actuation delay as the time taken from the dispatch of a command at the controller to the corresponding actuation by a drive. Similarly, the actuation deviation is defined as the time difference between the earliest and latest actuation at different drives in response to the commands of the same cycle. Therefore, in order to deliver the highest possible precision and accuracy, we should be able to optimize the tunable design paramters such as task periods so that the derived design minimizes the actuation delay and deviation.

For our analysis, we use the software model for EtherCAT servo drive that has been proposed by Kim *et al.* [Kim et al. 2012]. EtherCAT is one of the industrial Ethernet standards [Benzi et al. 2005, Jansen and Buttner 2004, Sung et al. 2013, Robert et al. 2012]. It is widely used for precision automation because it has numerous desirable features such as short message delivery time, as low as dozens of microseconds and globally synchronized clock with jitter in the sub-microsecond range. In an EtherCAT-based motion system, the motion controller transmits real-time messages with control variables such as target position, actual position, and control mode. The variables are defined as PDOs (Process Data Objects) in accordance with the CANopen drive profile [CANopen]. Using dedicated switch hardware, each drive relays the messages to the next one. When a message is relayed in the forward path, the output and input data in the message is, respectively, written to and read from the memory in the drive. Once the message frame arrives at the end of the network, it returns to the controller. Although our analysis assumes EtherCAT as the communication link, it is general such that, with minor modification of the communication submodel, it can be easily extended to other drive software using different network technology.

We base our drive model on the multitasked software implementation with a lightweight real-time kernel [Kim et al. 2012]. To achieve better responsiveness, it uses two-level preemptive scheduling where all periodic tasks are classified into two groups according to their priorities, and each group has its own scheduler. The scheduler task for the lower-priority group can never be executed when a task belonging to the higher-priority group is executing, thus the tasks in the higher-priority group do not suffer from the scheduling overhead by the lower-priority tasks. This mechanism is implemented using semaphore operations, *i.e.*, *sem_pend* and *sem_post*. When the scheduler task wants to release a task, it calls *sem_post*, waking up the task pending on the corre-

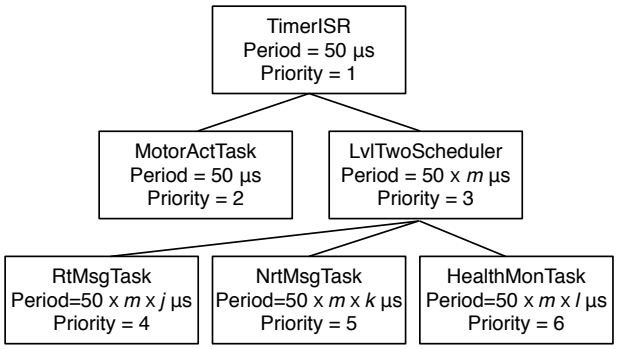


Figure 3: Multitasked drive software with two-level scheduling [Kim et al. 2012].

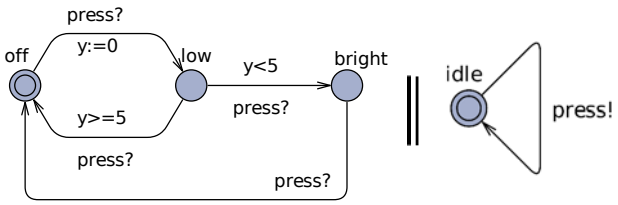
sponding semaphore.

Figure 3 shows the organization of the drive tasks with their priorities and requirements on periods. The higher-priority group is composed of MotorActTask and LvlTwoScheduler tasks. The MotorActTask performs motor actuation and sensing according to the commands from the motion controller. Since MotorActTask has the most stringent timing requirement, it is assigned the highest priority next to the first-level scheduler, the timer interrupt service routine (TimerISR). The TimerISR has a period of 50 μs, which has been fixed according to the requirement from the drive hardware.

The LvlTwoScheduler is responsible for scheduling the tasks in the lower-priority group, which includes RtMsgTask, NrtMsgTask, and HealthMonTask. The priority of LvlTwoScheduler should be lower than MotorActTask but higher than any of tasks in the lower-priority group. So, its period is set to be the greatest common divisor of the task periods in the scheduling group. The RtMsgTask implements the protocol stack for EtherCAT and CANopen. This task extracts the control commands from the real-time EtherCAT message and stores them into a memory area shared with MotorActTask. The RtMsgTask also reports motor status such as current position, speed, and torque information to the motion controller. The HealthMonTask monitors the status of the motor drive, such as voltage and current. The NrtMsgTask handles non-real-time EtherCAT messages, which may contain system information gathered by HealthMonTask.

2.2 Timed Automata

Timed automata is an extended finite-state machine with real-valued clocks [Alur and Dill 1994]. A timed automaton is a tuple (L, l_0, C, A, T, I) where L is a finite set of locations, l_0 is the initial location, C is the set of clocks, A is a set of actions, T is a set of transitions, and I assigns invariants to locations. A transition is a tuple (l_1, a, b, c, l_2), where l_1 is a source location, a is a synchronization action, b is a boolean expression over clocks, c is a set of reset clocks, and l_2 is a target location. In timed automata, a system S is modeled as a composition of processes P_1, P_2, ..., P_n



(a) Lamp (b) User

Figure 4: Timed automata of a simple lamp [UPPAAL].

that are defined as timed automata, expressed by $S = P_1||P_2||...||P_n$. The processes run in parallel, and can be synchronized through channels. For a channel ch, a process that performs an output action ch! is synchronized with another process that performs an input action ch?.

Figure 4 shows a timed automata model for a simple lamp in [UPPAAL]. The process Lamp has three locations: off, low, bright. It is initially at location off, and can move to location low by executing press?. With the transition, clock y is reset to zero. At location low, the clock y starts to increase. If it executes press? before 5 time units (y<5), it moves to location bright. Otherwise, it returns to the initial location. The process User can execute press! at any time. The entire system is defined as SYSTEM = Lamp || User. The two processes must be synchronized with channel press.

Timed automata in UPPAAL are extended with additional features: constants, bounded integer variables, stopwatches, urgent channels, committed locations, arrays, user functions, and so on. UPPAAL provides a model-checker to verify a timed automata model with respect to a given requirement which is specified in a simplified version of TCTL (Timed Computation Tree Logic). Using TCTL, we can specify various properties such as reachability, safety, and liveness. In TCTL, $A\Box p$ means that p is always true for all traces, and $E\Diamond p$ means that p is eventually true for some trace. UPPAAL also provides a simulator that is used for the user to run the system manually. Moreover, we can go through a counter-example trace given by the model-checker to see how certain states are reachable.

2.3 Related Works

Formal verification based on timed automata has been used in various time-critical and safety-critical application domains, such as nuclear engineering, satellite systems, medical systems, and industrial automation. Lahtinen *et al.* [Lahtinen et al. 2012] demonstrated the efficiency of model checking in the system-level analysis for real-world industrial systems such as stepwise shutdown system and uninterruptible power supply (UPS) control software. Pajic *et al.* [Pajic et al. 2012] used a model-driven approach that combines Simulink and UPPAAL to analyze the safety of the closed-loop medical system, a PCA infusion pump. Mokadem *et al.* [Mokadem et al. 2010] used timed

automata and UPPAAL to specify and verify the functional correctness and timing requirements of a commercial automation system implemented in multi-tasked PLC program. Ruel *et al.* [Ruel et al. 2009] presented an iterative model checking approach to find the bounds of response time in a networked automation system.

With the explosive growth of software complexity in industrial systems, the timed automata model and formal verification of multitasking applications running under real-time OS has also received substantial attention. Waszniowski *et al.* [Waszniowski and Hanzálek 2008] presented timed automata models of multiple application tasks, OSEK compliant OS kernel, and an ISR. With the automated gearbox control system, they analyzed the worst-case response time (WCRT) of the tasks through model checking. They first defined the timing property of a task using TCTL formula in UPPAAL. And, they explored the WCRT of tasks by setting the initial values of the WCRT to those estimated by the designer and iteratively decreasing the value until the TCTL formula is not satisfied. Mikucionis *et al.* [Mikučionis et al. 2010] proposed a modeling framework using UPPAAL to perform the schedulability analysis of a multi-tasked satellite system including block factor and CPU utilization. They showed that the model-based approach provides a safe but less pessimistic result compared with classical scheduling theory based approach.

The end-to-end delay in networked motion control systems is one of the important performance metrics, and thus some previous studies have addressed it. Early works formulated the end-to-end delay of the EtherCAT network and presented the achievable Minimum Cycle Time (MCT) according to the varying slave numbers and packet size [Prytz 2008, Jasperneite et al. 2007]. Seno *et al.* [Seno and Zunino 2008] also analyzed the MCT of EtherCAT-based control system by extensive simulation. However, these studies did not consider the device-level delay factors, only dealing with the network transmission delay. Recent studies have started to address the impact of the motion controller or the internal operation of motor drives on the end-to-end delay. Cereia *et al.* [Cereia et al. 2011] evaluated the performance of the Linux-based controller in terms of the cycle accuracy of periodic control task by measurement. Kim *et al.* [Kim et al. 2012] presented a combination of the minimized periods based on the various sets of the required deadline miss probabilities of the tasks and analyzed the end-to-end delay on the EtherCAT drives.

3 Modeling of Motion Control System

In this section, we present a timed automata model for the motion control system described in Section 2.1. As shown in Figure 2, the entire system is defined as SYSTEM which is composed of a motion controller MotionController, one or more motor drives $\mathsf{MotorDrive}_i$ and links Link_i.

$$\mathsf{SYSTEM} = \mathsf{MotionController} \parallel \mathsf{MotorDrive}_1 \parallel \cdots \parallel \mathsf{MotorDrive}_n \parallel \mathsf{Link}_1 \parallel \cdots \parallel \mathsf{Link}_n$$

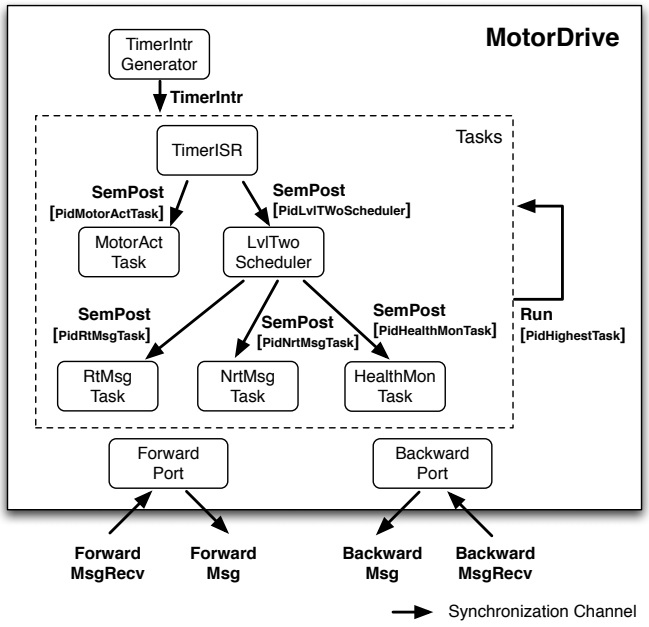


Figure 5: Components and channels of a motor drive.

The motion controller consists of process ControlTask which periodically generates control commands, and processes TxPort and RxPort which act as the EtherCAT communication ports.

$$\mathsf{MotionController} = \mathsf{ControlTask} \parallel \mathsf{TxPort} \parallel \mathsf{RxPort}$$

The control task ControlTask receives state information from the motor drives, calculates the next control command, and sends the command via the EtherCAT ports for each control task's cycle. Process TxPort transmits commands from ControlTask to the EtherCAT link. On the other hand, process RxPort receives messages from the EtherCAT link and forwards them to ControlTask.

Depending on the full-duplex mechanisms, an EtherCAT Link is modeled as a forward link ForwardLink and a backward link BackwardLink as follows.

$$\mathsf{Link}_i = \mathsf{ForwardLink}_i \parallel \mathsf{BackwardLink}_i$$

The processes ForwardLink and BackwardLink just transfer incoming messages to their adjacent motor drives.

Figure 5 shows the components and synchronization channels in the motor drive process. The process MotorDrive is defined as follows.

$$\begin{aligned}\mathsf{MotorDrive}_i = {} & \mathsf{TimerIntrGenerator}_i \parallel \mathsf{TimerISR}_i \parallel \mathsf{MotorActTask}_i \\ & \parallel \mathsf{LvlTwoScheduler}_i \parallel \mathsf{RtMsgTask}_i \parallel \mathsf{NrtMsgTask}_i \parallel \mathsf{HealthMonTask}_i \\ & \parallel \mathsf{ForwardPort}_i \parallel \mathsf{BackwardPort}_i\end{aligned}$$

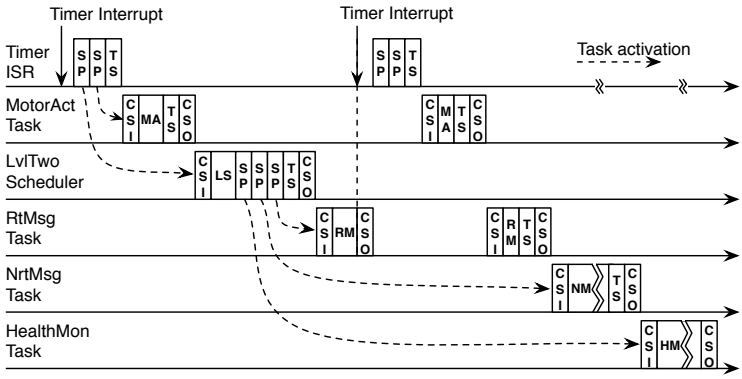


Figure 6: Components of the task execution times [Kim et al. 2012].

Process TimerIntrGenerator periodically generates a timer interrupt. Two processes TimerISR and LvlTwoScheduler play roles as schedulers, and the next four processes MotorActTask, RtMsgTask, NrtMsgTask, and HealthMonTask perform their own works. Process ForwardPort transfers messages from the motion controller or its previous drive to the next drive, and process BackwardPort transfers the returning messages. There are three kinds of synchronization channels. First, channel TimerIntr indicates the timer interrupt that preempts the currently running task. Second, channel SemPost[pid] represents the semaphore post operation, and the task with pid changes its status to Ready if a synchronization occurs through the channel. The first level scheduler TimerISR inserts MotorActTask and LvlTwoScheduler in the higher-priority group to the ready queue through events SemPost[PidMotorActTask]! and SemPost[PidLvlTwoScheduler]!, respectively. Similarly the second level scheduler LvlTwoScheduler enqueues RtMsgTask, NrtMsgTask, and HealthMonTask in the lower-priority group through the corresponding events. Third, channel Run[PidHighestTask] represents the wake-up call from the currently running task to the highest priority task in the ready queue. PidHighestTask indicates the pid of the task with the highest priority in the ready queue. Whenever a task is finished, it wakes up the next task through event Run[pidHighestTask]!.

Figure 6 illustrates how the task execution times are accounted for. Each task requires time for one context switch. Thus, one context switch in (CSI) and one context switch out (CSO) are added to the pure execution time of each task. Moreover, the task scheduling overhead (TS), which occurs when the task transfers the control to a lower priority task designated, is added to the pure execution time of each task. And if the task plays the role of the scheduler, the semaphore operation overhead (SP), which occurs when the scheduler task wants to release a task in the lower-priority group designated by the decentralized scheduling framework, is also added. Note that TimerISR does not have context switch overhead because it does not need to maintain its context. In the figure, the pure execution times of tasks are represented by MA (MotorActTask), LS (LvlTwoScheduler), RM (RtMsgTask), NM (NrtMsgTask),

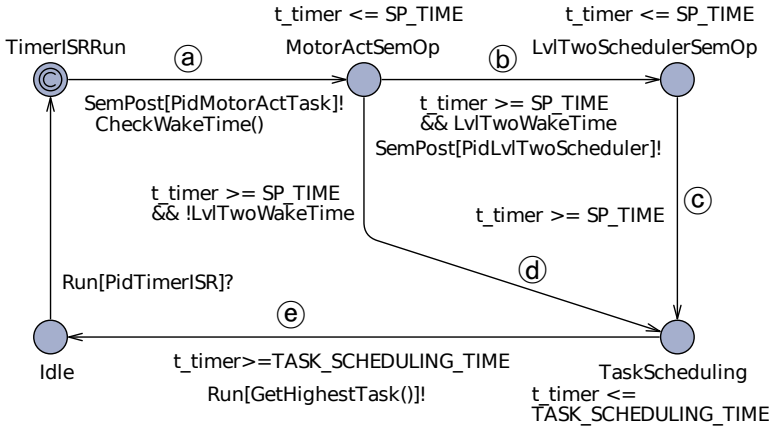


Figure 7: Timed automata model of TimerISR.

and HM (HealthMonTask).

Now, we define the components in timed automata. In this paper, we omit less important information (*e.g.*, clock and variable assignments) on figures of timed automata because the figures are too complex to understand if all the details are included. First of all, TimerIntrGenerator periodically generates a timer interrupt in order to wake TimerISR. The process includes two locations Idle and Interrupted. At location Idle, it waits for its period (*e.g.*, 50 μs), and then moves to location Interrupted. With this transition, it sends the synchronization event TimerIntr! to halt the currently running task. At location Interrupted, it immediately comes back to the Idle location for its next period.

The TimerISR process for an interrupt service routine plays the role of the first level scheduler in the motor drive. Listing 1 shows the pseudo code for TimerISR. It wakes up MotorActTask and LvlTwoScheduler properly using semaphore functions. The corresponding timed automata model is given in Figure 7. The TimerISR process initially stays at location Idle, and moves to location TimerISRRun if it receives an event through channel Run[PidTimerISR]. The period of TimerISR is the same as the period of TimerIntrGenerator (*i.e.*, the TimerIntr period) because the currently running task interrupted by TimerIntr immediately releases Run[PidTimerISR]!. At the TimerISRRun location, the process performs the semaphore post operations for MotorActTask and LvlTwoScheduler. It always executes SemPost[PidMotorActTask]! because the periods of MotorActTask and TimerISR are the same. In contrast, the execution of SemPost[PidLvlTwoScheduler]! depends on the result of function CheckWakeTime() at transition ⓐ in Figure 7. The function checks whether a period of LvlTwoScheduler is over or not. Depending on the result, it moves to location TaskScheduling through transition sequence ⓐⓑⓒ or the other transition sequence ⓐⓓ. Finally, it comes back to the Idle location by transition ⓔ. With the transition, it selects a task in the ready queue by using function GetHighestTask(), and wakes up the task through the Run channel. In fact, MotorActTask is always chosen

Listing 1: Pseudo code for TimerISR and MotorActTask.

```
semaphore MotorActSem;
semaphore LvlTwoSem;

isr TimerISR()
{
    sem_post(MotorActSem);
    if it is time to wake LvlTwoScheduler
        sem_post(LvlTwoSem);
    end if

    Determine the highest priority task τ in the ready queue;
    Switch to task τ;
}

task MotorActTask()
{
    while (TRUE)
        sem_pend(MotorActSem);
        Perform motor actuation and sensing;
    end while
}

os_primitive sem_post(semaphore Sem)
{
    if wait queue for Sem is empty
        Increment the counter for Sem;
        return;
    end if

    Make ready the highest priority task τ in the wait queue of Sem;
}

os_primitive sem_pend(semaphore Sem)
{
    if counter for Sem is greater than zero
        Decrement the counter;
        return;
    end if

    Add the calling task to the wait queue of Sem;
    Determine the highest priority task τ in the ready queue;
    Store the context of the calling task;
    Load the context of task τ;
}
```

because it is the highest priority task except for TimerISR.

The process MotorActTask, shown in Figure 8, is initially located at Waiting. If it receives an event from its scheduler TimerISR through channel SemPost[PidMotorActTask], it moves to Ready location. When this transition happens, it is inserted to the ready queue by function InsertReadyQueue(). At the Ready location, it waits for an event coming from channel Run[PidMotorActTask] and enters its actual execution phase. The phase consists of ContextSwitchIn, Running, TaskScheduling, and ContextSwitchOut. Through the phase, it stays at each location for the given time and moves to its next location,

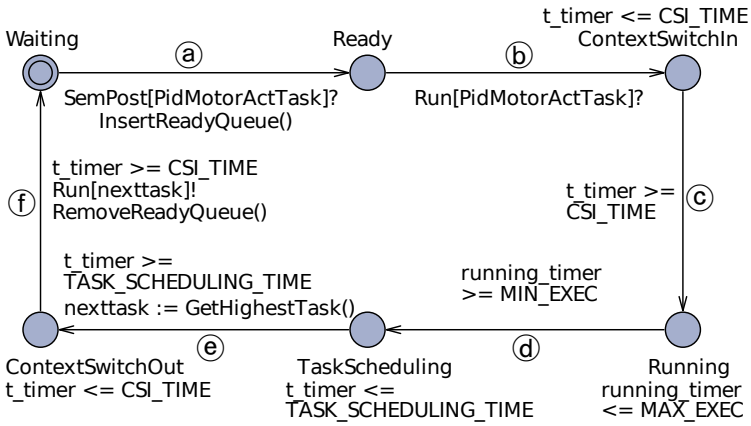


Figure 8: Timed automata model of MotorActTask.

as described in transitions ⓒ,ⓓ,ⓔ,ⓕ. With the transitions ⓔ and ⓕ, it selects the next task via GetHighestTask() function and releases Run[nexttask] event, dequeues the task from the ready queue, and then comes back to the Waiting location.

The timed automata model of the LvlTwoScheduler process that plays the role of the second level scheduler is an extension of the model of MotorActTask with semaphore post operations for lower priority tasks and interrupt handling operations for the timer interrupt. For LvlTwoScheduler, the location Running in Figure 8 is extended to four locations LvlTwoRunning, RtMsgSemOp, NrtMsgSemOp and HealthMonSemOp, as shown in Figure 9. At the transition from ContextSwitchIn to LvlTwoRunning, the process checks whether periods of three lower priority tasks expire via the CheckWakeTime() function. According to the result, LvlTwoScheduler optionally moves step by step to RtMsgSemOp, NrtMsgSemOp, or HealthMonSemOp like transitions ⓐ. That is, LvlTwoScheduler may release synchronization events SemPost[PidRtMsgTask], SemPost[PidNrtMsgTask], and SemPost[PidHealthMonTask] depending on the expiration of their periods. Since LvlTwoScheduler is interruptible, it moves to location ContextSwitchOut with TimerIntr synchronization as soon as a timer interrupt occurs, which is described by transitions ⓑ. After the transitions, LvlTwoScheduler makes TimerISR run through Run[PidTimerISR] channel and move to location Ready. When the process is restarted, it needs to be executed for its remaining time, and thus we add a stopwatch r_timer which is increased only in the LvlTwoRunning location.

Since the behaviors of RtMsgTask, NrtMsgTask, and HealthMonTask are similar, they can be defined as the timed automata model given in Figure 10. The automaton is an extension of MotorActTask with interrupt handling operations for the timer interrupt.

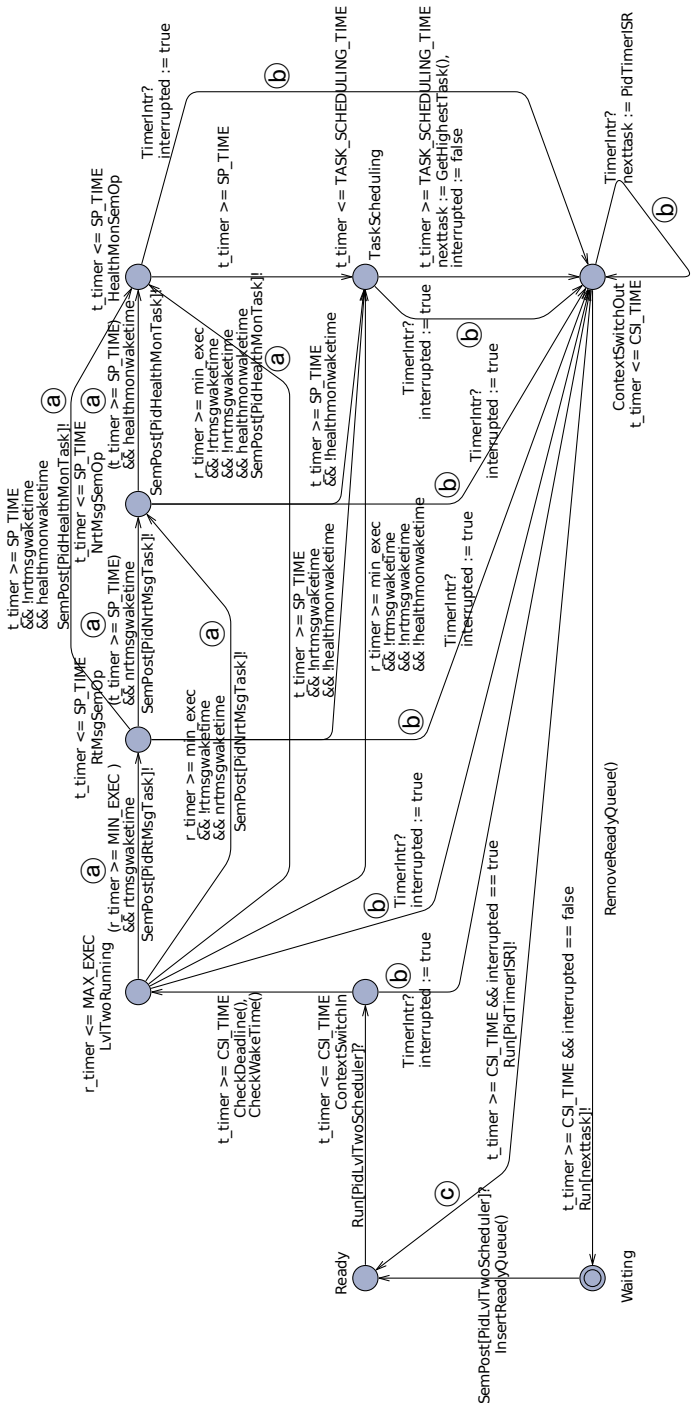


Figure 9: Timed automata model of LvlTwoScheduler.

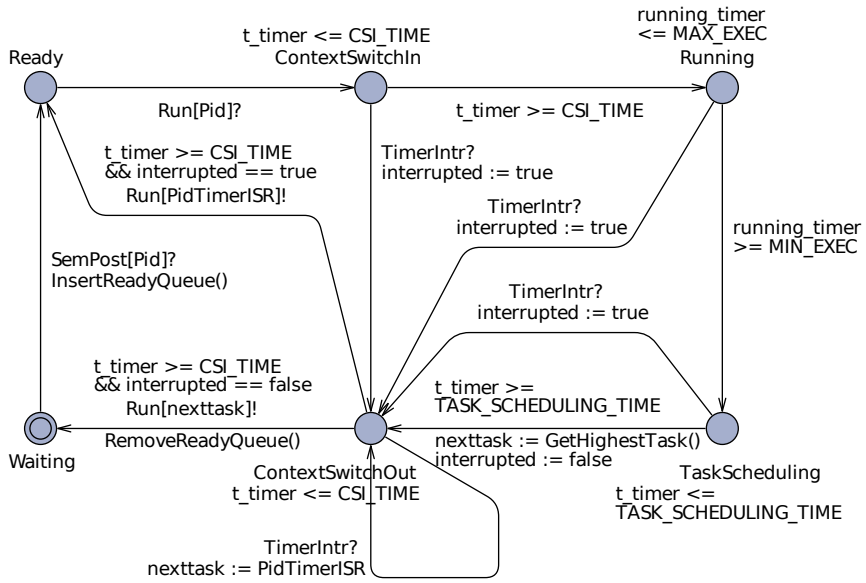


Figure 10: Timed automata model of RtMstTask, NrtMsgTask, and HealthMonTask.

In SYSTEM, ForwardPort and BackwardPort are in charge of transmitting and receiving EtherCAT messages for the motor drive, and are synchronized with ForwardLink and BackwardLink, respectively.

4 Analysis of Timing Requirements for Motor Drive Software

In this paper, we verify the timing requirements of motor drive software such as deadlock-freeness, schedulability, end-to-end delay requirements, and actuation deviation requirements using the UPPAAL model checker. We define a boolean function $modelcheck(p_{TimerISR}, p_{LvlTwoScheduler}, TCTLformula)$ to be true if and only if the model checker outputs *"Property is satisfied"* for the formula with the given periods. Here, we define p_A as the period of task A. We note that the periods of TimerIntrGenerator, TimerISR and MotorActTask are the same, and the period of LvlTwoScheduler is multiple of that of TimerISR and is same as the period of RtMsgTask.

The constants SP_TIME, CSI_TIME, CSO_TIME, MIN_EXEC, MAX_EXEC, and TASK_SCHEDULING_TIME in the timed-automata model represent the execution times for the corresponding locations. In this paper, the values come from experimental results in Kim *et al* [Kim et al. 2012], as shown in Table 1. We note that the values depend on the types of tasks and the size of PDO. For example, in the case of 24-byte PDO set, MIN_EXEC and MAX_EXEC of RtMsgTask in Figure 10 are 14.60 and 14.67 μs, respectively.

Table 1: The values of constants [Kim et al. 2012].

Constants		Values (Unit : μs)
commons	CSI_TIME	0.87
	CSO_TIME	0.87
	TASK_SCHEDULING_TIME	0.73
TimerISR, LvlTwoScheduler	SP_TIME	3.60
MotorActTask	MIN_EXEC	18.00
	MAX_EXEC	35.26
LvlTwoScheduler	MIN_EXEC	4.77
	MAX_EXEC	4.77
RtMsgTask (24-byte PDO set)	MIN_EXEC	14.60
	MAX_EXEC	14.67
RtMsgTask (100-byte PDO set)	MIN_EXEC	45.63
	MAX_EXEC	45.70
NrtMsgTask	MIN_EXEC	9.47
	MAX_EXEC	9.63
HealthMonTask	MIN_EXEC	8.50
	MAX_EXEC	8.50

4.1 Verification of Deadlock Freeness

For safety, we must ensure the deadlock freeness in systems. We can verify the absence of deadlock by UPPAAL model checker with the following TCTL formula.

$$f_{safe} = \mathsf{A}\Box(\neg\ \mathsf{deadlock}) \tag{1}$$

First of all, we set the period of the timer interrupt service routine TimerISR to 50 μs and the period of the second level scheduler LvlTwoScheduler to 300 μs, as given in [Kim et al. 2012]. Recall that the TimerISR period is fixed due to the requirement of the drive hardware used in our study. Then, the model checker shows the output *"Property is satisfied"*, that is, modelcheck(50, 300, f_{safe}) is true.

Next, we determine the minimum period of LvlTwoScheduler for a deadlock-free system using the following algorithm, where $p_{TimerISR}$ is the period of TimerISR.

Listing 2: Minimum period of LvlTwoScheduler.

```
1 satisfied = false;
2 for ( k = 1; not satisfied; k++)
3       p = p_TimerISR * k;
4       satisfied = modelcheck(p_TimerISR, p, f_safe);
5 min_LvlTwoScheduler = p;
```

When $p_{TimerISR}$ is 50 μs, the minimum period of LvlTwoScheduler is 150 μs. The system remains free of deadlock for the rest periods greater than the found value. It is guaranteed because we have the constraint that task periods should be multiple of $p_{TimerISR}$. The synthesis of task periods in general cases is a complicated problem, which is beyond the scope of the paper [Davare et al. 2007]. The result informs developers that

LvlTwoScheduler should be assigned its period to larger than or equal to 150 μs. In other words, the period of LvlTwoScheduler should be at least three times the period of TimerISR for the development of deadlock-free systems.

The shorter period of timer interrupt makes it possible to develop the more precise system. Therefore, we present how to obtain the theoretical minimum period of TimerIntrGenerator for deadlock-free systems. We repeatedly reduce the period by 0.01 μs starting from 50 μs and perform model checking until deadlock occurs as follows.

Listing 3: Theoretical minimum period of TimerIntrGenerator.

```
satisfied = true;
for (p = 50; satisfied; p = p - 0.01)
     satisfied = modelcheck(p, p*3, f_safe);
min_TimerIntrGenerator = p + 0.01;
```

The theoretical minimum of TimerIntrGenerator's period is 46.53 μs. This means that the system is deadlock-free if the period is larger than or equal to 46.53 μs. For an unsafe case, we examine the trace that the model checker produces as a counter example, and find the case that TimerIntrGenerator releases TimerIntr before the execution of MotorActTask is completed.

4.2 Schedulability Analysis

In real-time systems with periodic tasks, it is important to check whether the periodic tasks are schedulable, *i.e.,* whether the tasks are completed in their deadline. For the motion control system, we must guarantee that RtMsgTask does not miss the deadline since it deals with hard real-time messages. Thus, we present how to analyze schedulability of the hard real-time task RtMsgTask.

We add a boolean variable rt_miss to the scheduler LvlTwoScheduler in Figure 9. The rt_miss variable is used to indicate whether or not RtMsgTask misses its deadline. The transition from ContextSwitchIn to LvlTwoRunning includes a new function CheckDeadline() in addition to CheckWakeTime(). Via CheckWakeTime(), the scheduler determines whether to start a new period at this time for each task. If the previous work of the task is not yet completed, it is still placed in the ready queue. In other words, if a newly scheduled task exists in the ready queue, we can decide that this task does not meet its deadline. Therefore, function CheckDeadline() assigns the corresponding boolean variable to true if the task is in the queue. We can analyze the system's schedulability using the following TCTL formulas.

$$f_{schedulable} = \mathrm{A}\Box(\neg\ \mathrm{LvlTwoScheduler.rt\_miss}) \tag{2}$$

The following algorithm outputs the minimum value of RtMsgTask's period. As mentioned, the period of RtMsgTask is same as the period of LvlTwoScheduler to minimize the RtMsgTask's response time. Also, as the result of Listing 2, the period of LvlTwoScheduler should be at least three times of that of TimerISR, and thus we start from k=3.

Listing 4: Minimum period of RtMsgTask.

```
satisfied = false;
for (k = 3; not satisfied; k++)
     p = p_TimerISR * k;
     satisfied = modelcheck(p_TimerISR, p, f_schedulable);
min_RtMsgTask = p;
```

For the 24-byte PDO set, the result shows that RtMsgTask is always schedulable if the period of RtMsgTask is greater than or equal to 300 μs. And, for the 100-byte PDO set, RtMsgTask satisfies its deadline if the period is not less than 550 μs. The feasible periods for 100-byte PDO are larger than the period for 24-byte PDO because of the difference of the execution times. Using this approach, we can decide the periods of real-time tasks at the design stage.

4.3 Motion Precision and Accuracy Analysis

One of the fundamental timing requirements of networked motion systems is the requirement on the end-to-end actuation delay. It is a metric of motion precision because the shorter end-to-end delay makes higher precision of single axis control. Another essential timing requirement is on the actuation deviation that is an indicator of accuracy. The smaller actuation deviation increases the accuracy of multi-axis motion. In this section, we apply the UPPAAL model-checker to prove the timing requirements on the end-to-end delay and actuation deviation as well as to find the minimum and maximum of them.

First of all, we derive a drive-local delay before analyzing the timing requirements. The drive-local delay is defined as the time taken from message reception to motor actuation in a motor drive. For the drive-local delay, we provide an additional timed automata TimeMeasureTask in Figure 11. Initially, TimeMeasureTask waits for a message from the motion controller, and transits from Idle to MsgReceived as soon as the message arrives at ForwardPort. With the transition, a clock timer is reset to zero. The TimeMeasureTask process moves to the next location whenever the following events occurs: start of RtMsgTask, completion of RtMsgTask, start of MotorActTask, and completion of the MotorActTask process. Finally, TimeMeasureTask arrives at location Actuation. Since the location is a committed location, the process immediately moves to Idle. Therefore, the value of timer at the Actuation location is the drive-local delay. We note that the drive-local delay depends on the periods of RtMsgTask and MotorActTask.

If the timing requirement for the drive-local delay is given by [*MIN, MAX*], we can express the requirement as the following TCTL formula.

$$f_{delay} = \mathrm{A}\Box(\mathrm{TimeMeasureTask.Actuation}) \rightarrow (\mathrm{TimeMeasureTask.timer} \geq \mathrm{MIN}) \wedge (\mathrm{TimeMeasureTask.timer} \leq \mathrm{MAX}) \quad (3)$$

We can check whether this formula is satisfied for MotorDrive using the UPPAAL model checker. However, designers may not know the timing requirement on bounds of

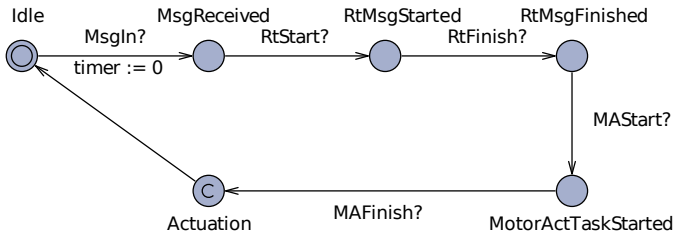


Figure 11: Timed automata model of TimeMeasureTask.

the drive-local delay. Thus we propose a method of deriving the minimum and maximum values of the drive-local delay.

Let us consider the following TCTL formula $f_{max}(t)$.

$$f_{max}(t) = \mathsf{E}\Diamond(\mathsf{TimeMeasureTask.Actuation}) \wedge (\mathsf{TimeMeasureTask.timer} \geq \mathsf{t}) \qquad (4)$$

The formula $f_{max}(t)$ is satisfied if and only if there exists a trace which reaches the state such that TimeMeasureTask is at Actuation and the value of timer is larger than or equal to t. Thus, if this formula is false, the value of timer is always smaller than t at Actuation. On the contrary, if this formula is true, timer can have a value larger than or equal to t. For some k, if $f_{max}(k+1)$ is false and $f_{max}(k)$ is true, we conclude that k is the maximum. To find the maximum, we start with a drive-local delay's upper bound u which is specified by system requirements. We repeatedly model-check the formula by decreasing k by 0.1 until $f_{max}(k)$ becomes true.

Listing 5: Maximum value of drive local delay.

```
1 select an upper bound u which is specified by system requirements;
2 satisfied = false;
3 for (k = u; not satisfied; k=k-0.1)
4      satisfied = modelcheck(p_TimerISR, p_LvlTwoScheduler, f_max(k − 1));
5 MAX = k;
```

We use similar approach to find the minimum of the drive-local delay.

$$f_{min}(t) = \mathsf{E}\Diamond(\mathsf{TimeMeasureTask.Actuation}) \wedge (\mathsf{TimeMeasureTask.timer} \leq \mathsf{t}) \qquad (5)$$

For some k, if $f_{min}(k-1)$ is false and $f_{min}(k)$ is true, k is the minimum. To find the minimum, we start with a value 0. We repeatedly model-check the formula by increasing k by 0.1 until $f_{min}(k)$ becomes true.

Listing 6: Minimum value of drive local delay.

```
1 satisfied = false;
2 for (k = 0; not satisfied; k=k+0.1)
3      satisfied = modelcheck(p_TimerISR, p_LvlTwoScheduler, f_min(k + 1));
4 MIN = k;
```

The drive-local delay depends on the periods of MotorActTask (or TimerISR) and RtMsgTask (or LvlTwoScheduler). In this experiment, we set the period of TimerISR to 50 μs. Table 2 shows the maximum and minimum of the drive-local delay. The result gives the same minimum delay for the same PDO set. By analyzing traces, two tasks RtMsgTask and MotorActTask are not interrupted, and thus, the delay is the sum of minimum execution times of the tasks. In the case of the 24-byte PDO set, the drive-local delays are [37.8, 605.6] for period 300 μs and [37.8, 590.4] for period 350 μs.

End-to-end delay. The end-to-end delay is defined as the time from the dispatch of a command at the controller to the corresponding actuation. Therefore, this delay is the sum of the communication delay over links and the drive-local delay. Performance analysis for EtherCAT shows that the delay between adjacent devices is much less than 1 μs, and the end-to-end delay is proportional to the number of devices [Prytz 2008]. Suppose that the delay between adjacent drives is a constant c. For a command, the actuation occurs at time in $[Min+c, Max+c]$ at the first drive, and at time in $[Min+nc, Max+nc]$ for the last drive, where n is the number of drives. Then the end-to-end delay is given by $[Min+c, Max+nc]$. For example, the end-to-end delay is [77.8+16, 1105.6+16] with 100-byte PDO set and RtMsgTask's period of 550 μs for c=0.5 μs, n=32.

So far, the end-to-end delay is obtained from a motor drive, not from the entire system. From now on, we present how to verify the delay requirement from the system including the motion controller. Due to state explosion, we abstract SYSTEM as MotionController || Links || MotorDrive, where we model Links so that it takes $[c, nc]$ for the delivery. We also add a transition from Idle to CommandIssued and a transition from CommandIssued to MsgReceived instead of the transition from Idle to MsgReceived, and reset the clock timer to zero on the first transition in TimeMeasureTask in Figure 11. Finally, we can model-check whether SYSTEM satisfies the TCTL formula (5). As an experiment, we assign the period of ControlTask in MotionController as 2 *ms* and the period of RtMsgTask as 550 μs. Let c=0.5 μs, n=32. Then, we can prove the validity of the property for [93.8, 1121.6]. Moreover, more precise bound [123.2, 1044.8] is acquired by repeated experiments.

Actuation deviation. The actuation deviation is defined as the time difference between the earliest and latest actuation at different drives in response to the commands belonging to the same controller cycle. Thus the worst-case deviation is the difference from the maximum of the last drive's actuation to the minimum of the first drive's, that is $Max-Min+(n-1)*c$. Table 2 displays results on the actuation deviation for c=0.5 μs, n=32.

5 Conclusion

In this paper, we have presented a formal design approach for motion control systems. We describe a timed automata model for an Ethernet-based motion system, which is

Table 2: Analysis of drive-local delay and actuation deviation.

PDO set	Task period (μs)	Drive-Local Delay (μs)		Actuation
	RtMsgTask	Min	Max	Deviation (μs)
24-byte PDO	300	37.8	605.6	583.3
	350	37.8	590.4	568.1
	400	37.8	640.4	618.1
100-byte PDO	550	77.8	1105.6	1043.3
	600	77.8	1090.4	1028.1
	650	77.8	1140.4	1078.1

composed of a motion controller, motor drives, and communication links. For the precision and accuracy analysis of the drive software design, in particular, we model the motor drive in enough detail to accurately capture the task scheduling mechanism of a real drive implementation. With the developed model, we verify the timing requirements such as deadlock-freeness and real-time schedulability. And, using UPPAAL model checker, we evaluate the precision and accuracy of the motion system through verification of the requirements on the end-to-end actuation delay at each drive and actuation deviation between different drives, respectively.

The verification results show that our model-based approach enables efficient design space exploration for motion system design. Through experiments, we have shown that, for varying number of drives and size of messages, we can successfully determine the system safety and derive the combination of minimum task periods that leads to the best precision and accuracy. We also see that interrupt period in the motor drive can be feasibly reduced to 46.53 μs.

In our future research, we will extend the motion controller model by considering the host operating system and motion controller software, and study the modeling and verification of application-level motion specifications. We also plan to extend our model to conduct a stochastic analysis of the minimum possible control cycle time and the responsiveness of soft real-time tasks in the motor drive software using UPPAAL statistical model checking (SMC).

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2004984) for Taehyoun Kim. Also, this work was supported by the 2013 sabbatical year research grant of the University of Seoul for Inhye Kang.

References

[Alur and Dill 1994] Alur, R., and Dill, D. L.: "A theory of timed automata,"; Theoretical computer science, 126, 2 (1994), 183-235.

[Benzi et al. 2005] Benzi, F., Buja, G. S., and Felser, M.: "Communication architectures for electrical drives,"; IEEE Trans. Industrial Informatics, 1, 1 (2005), 47-53.

[Bon and Dutilleul 2013] Bon, P., and Dutilleul, S. C.: "From a Solution Model to a B Model for Verification of Safety Properties,"; Journal of Universal Computer Science, 19, 1 (2013), 2-24.

[CANopen] CANopen, CAL-based Communication Profile for Industrial Systems. [Online]. Available : www.canopen.org

[Cereia et al. 2011] Cereia, M., Cibrario-Bertolotti, I., and Scanzio, S.: "Performance of a real-time EtherCAT master under Linux,"; IEEE Trans. Industrial Informatics, 7, 4 (2011), 679-687.

[Choi 2013] Choi, Y.: "Model checking an OSEK/VDX-based operating system for automobile safety analysis,"; IEICE Trans. Information and Systems, E96-D, 3 (2013), 735-738.

[Davare et al. 2007] Davare, A., Zhu, Q., and Marco, D. N.: "Period optimization for hard real-time distributed automotive systems,"; Proc. Conf. on Design Automation (DAC), (2007), 278-283.

[Jansen and Buttner 2004] Jansen, D. and Buttner, H.: "Real-time Ethernet: the EtherCAT solution,"; Computing and Control Engineering, 15, 1 (2004), 16-21.

[Jasperneite et al. 2007] Jasperneite, J., Schumacher, M., and Weber, K.: "Limits of increasing the performance of industrial Ethernet protocols,"; Proc. 12th IEEE Int. Conf. on Emerging Technol. Factory Autom., (2007), 17-24.

[Kim et al. 2012] Kim, K., Sung, M., and Jin, H.-W.: "Design and implementation of a delay-guaranteed motor drive for precision motion control,"; IEEE Trans. Industrial Informatics, 8, 2 (2012), 351-365.

[Lahtinen et al. 2012] Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., and Heljanko, K.: "Model checking of safety-critical software in the nuclear engineering domain,"; Reliability Engineering & System Safety, 105, (2012), 104-113.

[Mikučionis et al. 2010] Mikučionis, M., Larsen, K. G., Ramussen, J. I., Nielsen, B., Skou, A., Palm, S. U., Pedersen, J. S., and Hougaard, P.: "Schedulability analysis using Uppaal: Herschel-Planck case study,"; Leveraging Application of Formal Methods, Verification, and Validation Lecture Notes in Computer Science, 6416, (2010), 175-190.

[Min et al. 2013] Min, H.-S., Chung, S.-M., and Choi, J.-Y.: "Deriving System Behavior from UML State Machine Diagram : Applied to Missile Project,"; Journal of Universal Computer Science, 19, 1 (2013), 53-77.

[Mokadem et al. 2010] Mokadem, H. B., Berard, B., Gourcuff, V., Smet, O. De, and Roussel, J.-M.: "Verification of a timed multitask system with UPPAAL,"; IEEE Trans. Automation Science and Engineering, 7, 4 (2010), 921-932.

[Pajic et al. 2012] Pajic, M., Mangharam, R., Sokolsky, O., Arney, D., Goldman, J., and Lee, I.: "Model-driven safety analysis of closed-loop medical systems,"; to appear in IEEE Trans. Industrial Informatics.

[Prytz 2008] Prytz, G.: "A performance analysis of EtherCAT and PROFINET IRT,"; Proc. 13th IEEE Int. Conf. on Emerging Technol. Factory Autom., (2008), 408-415.

[Robert et al. 2012] J. Robert, J.-P. Georges, E. Rondeau, and T. Divoux: "Minimum Cycle Time Analysis of Ethernet-Based Real-Time Protocols,"; International Journal of Computer, Communications & Control, 7, 4 (2012), pp. 743–757.

[Ruel et al. 2009] Ruel, S., Smet, O. de, and Faure, J.-M.: "Finding the bounds of response time of networked automation systems by iterative proofs,"; Proc. IFAC Symp. on Information Control Problems in Manufacturing, (2009), 1365-1370.

[Seno and Zunino 2008] Seno, L., and Zunino, C.: "A simulation approach to a real-time Ethernet protocol: EtherCAT,"; Proc. 13th IEEE Int. Conf. on Emerging Technol. Factory Autom., (2008), 440-443.

[Sung et al. 2013] M. Sung, I. Kim, and T. Kim: "Toward a Holistic Delay Analysis of EtherCAT Synchronized Control Processes,"; International Journal of Computer, Communications & Control, 8, 4 (2013), pp. 608–621.

[UPPAAL] UPPAAL. [Online]. Available : http://uppaal.org

[Vitturi et al. 2011] Vitturi, S., Peretti, L., Seno, L., Zigliotto, M., and Zunino, C.: "Real-time Ethernet networks for motion control,"; Computer Standards & Interfaces, 33, 5 (2011), 465-476.

[Waszniowski and Hanzálek 2008] Waszniowski, L., and Hanzálek, Z.: "Formal verification of multitasking applications based on timed automata model,"; Real-Time Systems, 38, 1 (2008), 39-65.

[Woodcock et al. 2009] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J.: "Formal methods: Practice and experience,"; ACM Computing Surveys, 41, 4 (2009), 19:1-19:36.

[Yu et al. 2009] Yu, D., Hu, Y., Yu, X. W., Huang, Y., and Du, S.: "An open CNC system based on component technology,"; IEEE Trans. Automation Science and Engineering, 6, 2 (2009), 302-310.