Task Sequencing for Optimizing the Computation Cycle in a Timed Computation Model

Sebastian Fischmeister and Guido Menkhaus, University of Salzburg, Austria

Abstract

Recent developments in embedded control systems promote the timed computation model following the principles of logical execution time (LET). Resulting control applications are time deterministic, value deterministic, and their properties may be subject to formal verification against a mathematical model of the control design. However, the timed computation model introduces inefficiencies to computation cycles. As the LET of a real-time control task requires being greater than its worst-case execution time and computed values are always propagated at the end of the LET, actuator updates are unnecessarily delayed. This makes the control cycle less responsive.
In this paper, we present an approach that allows the definition of task sequences for a timed computation model implemented by the timing definition language (TDL). Task sequences help minimizing timing delays between sensor readings and actuator updates (e.g., in estimator-based control systems), managing startup and shutdown phases of control systems, and providing mechanisms for error-detection in fault-tolerant systems.

Keywords: real-time control systems, timed computation model

1 Introduction

Reactive systems continuously receive stimuli from sensors, measure the environment, and react on the environment by controlling actuator states [1, 2]. The systems must be able to sense and control the environment within a predefined time frame for which the environment dictates the exact length. Such systems can be found, for instance, in real-time process controls in manufacturing or embedded control systems in the automotive area.

The design of a control system typically involves creating a set of periodically executed tasks. The system needs to react quickly to stimuli from the environment and sequencing of tasks is of primary concern [3]. A sequence defines a logical relationship between tasks and the release event of one task depends upon the termination of its preceding task. This allows using resources in a certain order with data dependencies within the same period.

Programming languages, which implement the timed computation model such as TDL [4] and Giotto [5] gained more recognition recently [6, 7, 8]. The essential idea of such languages is time-triggered cyclic computation in which computing a task follows reading inputs (e.g., sensor data) and precedes updating actuators. The logical execution time (LET) and the worst-case execution time (WCET) describe the timing behavior of a task. The task is released at the beginning and terminated at the end of the LET. The release and the termination event are time-triggered. According to a scheduling scheme, the task starts after it has been released and completes its execution before the LET has elapsed. The end of the LET specifies the instance of time when the outputs of the computation of a task become available to other tasks or actuators, even if the output of a task is available earlier (e.g., at the end of the WCET).

TDL, for example, declares a set of modes, which define sets of task invocations and other activities that are executed periodically and in parallel. A task has a set of input ports and a set of output ports. A driver provides values for the input ports and drivers assign values to the output ports. Reading input ports and writing to output ports happens at the

beginning and the end of the LET. This is the reason, why task sequencing in TDL with data dependencies can only be arranged across task periods and not within the same period.
The concept of a LET for tasks introduces determinism to the control system. A system is said to be value deterministic, if a set of tasks produces the same sequence of outputs given the same sequence of inputs, even though the environment changes. If the system produces the same output at always the same time, it is time deterministic. These properties apply only, if each task is itself value and time deterministic and the composition of the tasks is schedulable [9]. The timed computation model favors system determinism and predictability over code efficiency and system reactivity [10]. This a drawback when applying this model to real-time control systems.
In this article, we introduce task sequencing using micro-tasks. Micro-tasks help minimizing inefficiencies in the timed computation model. They are useful in reducing timing delays between sensor readings and actuator updates (e.g., estimator-based control systems), managing startup and shutdown phases of control systems, and providing mechanisms for error-detection in fault-tolerant systems.
The remaining of the article is structured as follows: Section 2 presents the motivation of the work. Related work is discussed in Section 3. We introduce the concept of task sequencing and micro-tasks for a timed computation model in Section 4. The implementation and a case study is discussed in Section 5 and we conclude the article in Section 6 with a brief outline about our future work.

2 Motivation

When programming embedded control systems in the automotive, aerospace, and manufacturing domain, it is necessary not only to specify a task execution frequency, but also a time-based order of the execution of tasks. The order defines a sequence of tasks having a specific execution frequency. It allows for splitting the calculation of a task into a set of micro-tasks to

- specify a time-based causal order of tasks, which require to be executed one after the other, and
- execute drivers for sensor readings and actuator updates intertwined with the invocation of micro-tasks.

Intertwined Invocation. The intertwined invocation of micro-tasks with actuator updates can reduce the delay between reading sensor values and writing actuator updates in timed computation models. Such a delay has no effect in control systems in which the invocation period of the task that computes the control algorithm is comparable to the computation time of the control algorithm. However, usually the computation time required to calculate the control algorithm is small compared to the sampling period. The delay of updating an actuator only at the end of the LET can be severe in feedback control systems, where a long delay could be disastrous for stability [11].
The specification of a sequence of tasks is desirable when the state of a real-time image of a real-time entity needs to be estimated before it is used [12]. A real-time image is the current state of a real-time entity. A real-time entity is a controlled object. The estimation is necessary, if the real-time image has lost its validity due to the progression of time between the time of observation of the real-time entity and the time of usage of its real-time image. This delay between observation and usage can be caused by the time it takes to transport the observation of the real-time entity from the place of observation to the place of processing or it is due to a slow sampling frequency of the real-time entity. The delay between observation and usage introduces an error. However, the control algorithm can approximate the error and estimate the state of the real-time image. Usually state estimation requires building a model of the dynamics of the real-time entity. The model allows the computation of the state of the real-time entity for the interval in which the real-time image has lost its validity [13].

System Startup and Shutdown. System startup and shutdown requires a specific sequence of tasks to be executed. Adherence to this sequence ensures that the system does not enter an unsafe state. In this sequence, parallel execution of the tasks is not an option, since the results of these tasks usually depend on each other and default/initial values may be unattainable. System startup and shutdown are no activities that are executed periodically and cannot be expressed by languages that support the description of periodic tasks, only.

Error Detection. Error detection can be implemented using a system design that includes model redundancy [14]. Model redundancy has been found effective against random errors. It uses a known model that estimates the properties of a physical system. Error detection identifies deviations between the model and the physical system. Task sequences are necessary, since the model is computed after the physical system has been controlled and before sensors report the new state of the system.

3 Related Work

Work on programming languages for control systems with real-time constraints can be broadly divided into three categories [9]:

- The scheduled model assigns priorities to tasks. A scheduler decides on the sequence of the execution of tasks, to meet all task deadlines, based on the current priorities of the set of active tasks.
- The synchronous approach assumes that the underlying platform is fast enough to execute all tasks related to an event before the next event arrives and the set of tasks related to this event is released.
- The timed model defines a LET for each task with a release and a termination event for each task. Other tasks or activities operating on the environment can use the outputs of a task only after the termination event of the task has occurred.

We look at representatives of these categories and the concepts they use to provide task sequencing.

OSEKWorks [15] is a OSEK/VDX compliant real-time operating system [16]. It supports tasks and different scheduling strategies and implements the scheduled-model approach. In OSEK/VDX each task must end with calling *TerminateTask* or *ChainTask*. *TerminateTask* ends the calling task. *ChainTask* defines a succeeding task to the calling task. OSEK/VDX specifies the succeeding task to be released as soon as the calling task has been terminated. Yet, the standard does not fully specify the behavior of *ChainTask* with respect to the scheduler and task queues. This may result in non-deterministic behavior of the control application.

Esterel [17] and Lustre [18] are imperative synchronous programming languages for programming hardware and software controllers. Synchronous languages abstract away the time required to react to an input and computing an output. The computing platform is assumed to be fast enough to react to an event before the next event arrives. Conceptually, the zero-delay value propagation (outputs become available as inputs become available) allows for concurrency and determinism to coexist. Esterel implements task sequencing by passing control from one task instantaneously to the succeeding task. In a sequence such as *p ; q*, the task described by the statement `q` starts immediately after task `p` has terminated.

XGiotto [10] is an event-triggered programming language but bases its computation model on the same abstraction as Giotto. It introduces new programming constructs that allow for reacting to synchronous and asynchronous events. The construct `react{b} until [e]` defines a reaction block with a body *b* and an event *e* that terminates the tasks released within the reaction block *b*. A sequence of reaction blocks is processed sequentially. The body of a reaction block may declare a LET, determined by the time when the event *e* (e.g., a time-triggered event) occurs. The system scheduler executes the set of tasks in the body of the reaction block concurrently and it is impossible to assign a LET to a single task. However, defining only one task per reaction block circumvents this limitation.

MetaH[19] is a software-architecture specification language. It describes how the different elements of a system (e.g., source, hardware, and communication) are integrated to form the final application. The tool chain of MetaH includes tools for visual editing, schedule or partition modeling, and analyzing relevant properties such as the timing behavior. Utilizing this tool chain, MetaH generates glue code for the interaction of individual elements of the system and takes care of the annotated real-time properties. MetaH provides means for creating task sequences via the concept of undelayed communication. Undelayed communication is implemented via precedence-scheduling constraints. MetaH generates scheduling constraints such that the source process (information producer) is dispatched and finishes execution before the target process (information consumer) is released. Undelayed communication and thus task sequencing in MetaH has the following constraints: At least one of the processes must be aperiodic or both processes have harmonic periods or the period of one of the processes is an integer multiple of the other. MetaH limits undelayed communication to local nodes only, i.e., two communicating processes must be bound to the same processor.
Currently, TDL lacks programming constructs to describe a time-based order of the execution of sequences of tasks and actuator updates. Possible solutions to this problem have been discussed in [20].

4 Task Sequencing with Micro-Tasks in TDL

Conceptually, TDL bases on Giotto, but it incorporates extensions such as the concept of a module, improved language syntax, and clean-house implementations of the underlying infrastructure on several hardware platforms [4].
Listing 1 shows a TDL module fragment. A TDL module consists of one or more TDL modes. A mode consists of several, possibly concurrently executed task invocations and actuator updates. A TDL mode specifies the period, i.e., the length of one computation cycle (e.g., 1000 ms). A task period is determined with respect to the execution period of the mode to which the task is assigned (by dividing the invocation period of the mode by the task frequency). The specification of the task frequency is part of the task declaration in the mode. Listing 1 shows the mode $main$, which declares task `mt1` with the frequency of *once-per-period* (i.e., period of 1000 ms), task `mt2` with a frequency of *twice-per-period* (i.e., period of 500 ms), and an actuator update with a frequency of *once-per-period* (i.e., period of 1000 ms).

Listing 1. TDL module fragment.

```
task m1 [WCET=100 ms] {
  input int i;
  output int o;
  uses f1(i,o);
}

task m2 [WCET=200 ms] {
  input int i;
  output int o;
  uses f2(i,o);
}

start mode main [1000 ms] {
  task
    [1] m1 {i := s2};
    [2] m2 {i := s1};
    ...
  actuator
    [1] act := mt1.o;
}
```

4.1 Concept of Micro-Tasks

The underlying concept of micro-tasks is sequential element composition [21] (see Figure 1). A task is composed of a sequence of non-concurrent execution of micro-tasks and actuator updates. The task to which the micro-tasks are assigned to, is called parent task. A micro-task itself can be a parent task to other micro-tasks or so-called child tasks.
Figure 1 shows the conceptual structure of TDL including the extension for task sequences. The TDL language consists of instances of the entities[1]

[1]The complete TDL language includes additional entities, however, they are not relevant for this paper. The language report [4] provides a complete list.

Actuator, *Task*, and *Mode*. The new concept of tasks allows for *Tasks* and *Modes* to include other groupable elements. Instances of *Tasks* can now contain other *Tasks* (denoted as micro-tasks) and *Actuators*.

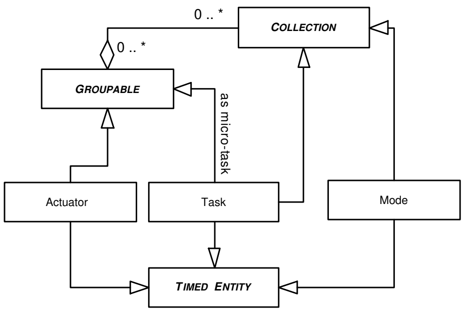

Figure 1. Structure of the TDL-language extension for micro-tasks.

Although both *Tasks* and *Modes* group elements such as tasks and actuators, they differ in syntax and treatment of the grouped elements.

- Instances of *Mode* consist of a set of tasks and actuators and declare their invocation period. Modes periodically schedule their set of tasks and actuators. Instances of *Task* only specify tasks and actuators. They do not specify frequencies for micro-tasks, which is rather determined by their parent task.

- An arbitrary declaration order of tasks and actuators in instances of *Mode* does not change the behavior of the control system. However, the correct declaration order of micro-tasks and actuators is key for the behavior of the system in instances of *Tasks*.

As a consequence of these changes, we can define sequences of tasks in TDL and have fine-grained access of execution paths.

4.2 Syntax

Listing 2 is an excerpt of a TDL program using micro-tasks.

Listing 2. TDL module fragment with micro-tasks.

```
task m1 [100 ms] {
  input int i;
  output int o;
  uses f1(i,o);
}

task m2 [200 ms] {
  input int i;
  output int o;
  uses f2(i,o);
}

task t {
  input int i;
  output int o;

  [LET = 200 ms] m1 {i := this.i};
  actuator act := m1.o;
  [LET = 300 ms] if g2() then
      m2 {i := m1.o};
}

start mode main [1000 ms] {
  task
    [1] t {i := s1};
    ....
}
```

Task `t` contains a sequence of two micro-tasks `mt1` and `mt2`. In between these two micro-tasks, it updates actuator `act`.
The syntax of this example differs from the standard TDL syntax. The basic declaration of a task is still compatible with the standard TDL syntax (see the two tasks `mt1` and `mt2`). The declaration of tasks has been extended allowing for sequences of tasks and actuator updates. Task `t` uses the extended syntax, as it consists of a sequence of task `mt1`, one actuator update, and, conditionally, task `mt2`.
To retain time and value determinism and be able to specify the exact timing of actuator updates, we introduce the attribute *LET*. This attribute specifies the LET of each micro-task. Actuators do not have a LET assigned, since they are executed synchronously in logical zero time. For example, in Listing 2, the LET of micro-task `mt1` is 200 ms (which is significantly larger than its WCET of 100 ms). Concluding from the LET of the micro-tasks and the order of instructions within the task `t`, the actuator will be updated 200 ms after the release of task `t`. The WCET of a parent task is computed from the LET of its micro-tasks.

4.3 E Code

The TDL compiler [4] compiles TDL source code and generates E code for the E machine [22]. The E machine is a virtual machine that executes the platform-independent E code and calls the platform-dependent functionality code. The E code ensures the timing consistency of the task executions (see Figure 2).

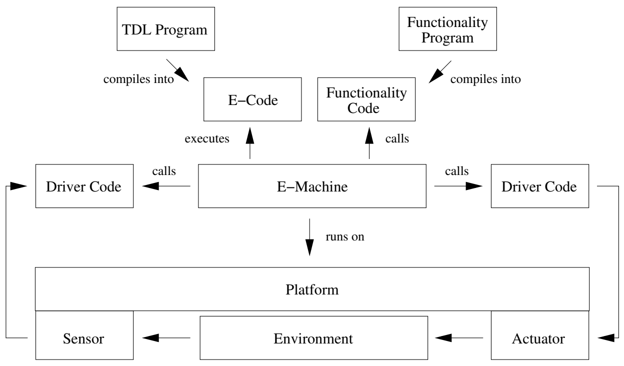

Figure 2. TDL system architecture.

Listing 3 presents a textual version of the E code for the TDL program of Listing 2.

Listing 3. The E code for the TDL program of Listing 2.

```
   lbl1 :
   call , d[t]
2           call , d[mt1]
            schedule , mt1
4           future , 200, lbl2
            return
6  lbl2 :   call , d[act]
            ifn , g2, lbl3
8           call , d[mt2]
            schedule , mt2
10 lbl3 :   future , 800, lbl1
            return
```

The `call` instruction synchronously executes drivers of a task. The `schedule` instruction releases a task to be activated by the scheduler of the operating system. The instruction `future` initiates the execution of a block E code instructions at some time in the future and the instruction `return` finishes the execution of the block of E code of the current label. For better readability, we added the instruction `ifn`, which is a conditional branch operation given the operand evaluates to false.

The E code is presented in assembler-like style: `lbl: opname(, arg1)?(, arg2)?`. Labels for branching operations are declared at the beginning of a line. The next argument lists the instruction name. The instruction name defines the operation, which the E machine executes. The following elements list arguments passed to the operation. Depending on the type of operation, we specify zero to at most two arguments.

The E code starts with the instruction at label `lbl1` (Line 1). In the first two instructions, the E machine calls the drivers of task `t` and micro-task `mt1`. This loads the arguments into the input ports of the tasks. The order of the driver calls is not arbitrary. As specified in the TDL example (see Listing 2), task `mt1` is a child-task of task `t` and requires as input the value of port `i` of task `t`. The driver of task `t` executes and terminates before the driver of micro-task `mt1`. The next instructions (Lines 3 to 5) prepare the micro-task `mt1` for dispatching, release a future condition, and hand over control to the scheduler. In the case study (Section 5), we provide more details about the E code.

Introducing micro-tasks requires additional blocks of E code. Each micro-task requires at least one additional label, additional drivers, and instructions (`schedule` and `future`). However, the additional E code is necessary to ensure determinism when using micro-tasks.

5 Case Study

In the following section, we present a case study of a control system utilizing micro-tasks for error detection.

5.1 Development Environment

The target platform of the research prototype of the E machine is the Motorola PowerPC 603 RISC processor. On this hardware, we run the Real-Time operating system for Embedded multi-processor Systems (RTEMS) [23, 24]. The selected compiler suite is the GNU compiler collection 3.2.3 [25] with

a set of patches specific to the target platform and the RTEMS operating system.
Using the PSIM application bundled with the GNU debugger suite [26], we simulate the Motorola PowerPC 603 RISC processor. Within this simulator, we run and debug PowerPC machine code that is equivalent to machine code that would be executed by a hardware processor. We chose the PowerPC platform for our research prototypes, for we later port the system to boards hosting a Motorola PC 555.
RTEMS is an operating system especially tailored to multi-processor systems and POSIX compliance. It is open-source, so we can tailor it to our needs. Similar to other real-time operating system, RTEMS provides basic services such as timers, scheduling, and communication. It includes a POSIX API that bases on the POSIX 1003.1b and supports a single process, multi-threaded environment. Regarding processes, RTEMS supports routines referred to as single user, single process.

5.2 Implementation Details of the E Machine

The E machine implementation in RTEMS uses the POSIX API to a large extent. Most functionality required by the E machine are standard services such as timers & clock, alarms, and scheduling. So, the implementation of the E machine can easily be ported to other POSIX-compliant platforms such as RT-Linux.
In the current design of the E machine research prototype, the E code is not statically linked to the E machine, the driver, and the functionality code. Instead, the E code is dynamically loaded at boot time. It uses the in-memory file system (IMFS) that RTEMS provides. The IMFS provides full POSIX filesystem functionality, yet it is RAM based. The E code is stored in the root directory of the IMFS. To provide the data at boot time, we build an image file of the IMFS that is statically linked to the `.data` section of the final executable. At boot time, RTEMS uses this image file to initialize the IMFS.
In the case study, we use the standard scheduler provided by RTEMS. The RTEMS scheduler allocates the processor using a priority-based, preemptive algorithm augmented to provide round-robin characteristics within individual priority groups. The goal of this algorithm is to guarantee that the task, which is executing on the processor at any point in time is the one with the highest priority among all tasks in the ready state [24].
Most instructions of the E machine are trivial. Instructions such as `ifn`, `call`, or `return` are implemented straightforwardly. We look into the implementation of the two instructions `schedule` and `future` in more detail. The instruction `schedule` dispatches a task. In the implementation, we initialize a data stack and create a new task using the POSIX call *pthread_create*. The new task is immediately dispatched. In the configuration of the RTEMS application, we must specify the maximum number of active tasks. Currently, this number is determined by the maximum count of schedule operations within one period of a mode. The instruction `future` is equivalent to a branch instruction that is evaluated at a specific point in time. The E machine implements this instruction via suspend operations and task preemption. The `future` instruction specifies the time span until the predicate associated to the `future` instruction evaluates to true. The E machine is suspended for this time span and wakes up at the specified point in time and preempts all tasks.

5.3 Splitting Control into Micro-Tasks

Defining task sequences is necessary, for example, for the design of controls laws using a current or predictive state estimation, when either the state elements of the system are inaccurate or they are not measured. It is possible to split the calculation for the estimation in a part before reading a sensor and a part after reading a sensor. In the case study in Section 5.4, we use a similar approach for model-based error detection.
Figure 3 shows the logical execution of a task `t` computing the estimator and the control law. The LET of task `t` and thus its invocation period equals T. When the task is released the estimator calculates state element estimations and the control law computes actuator update values. The actuator values are made accessible to the actuator only when the termination event of the task occurs, i.e., at the

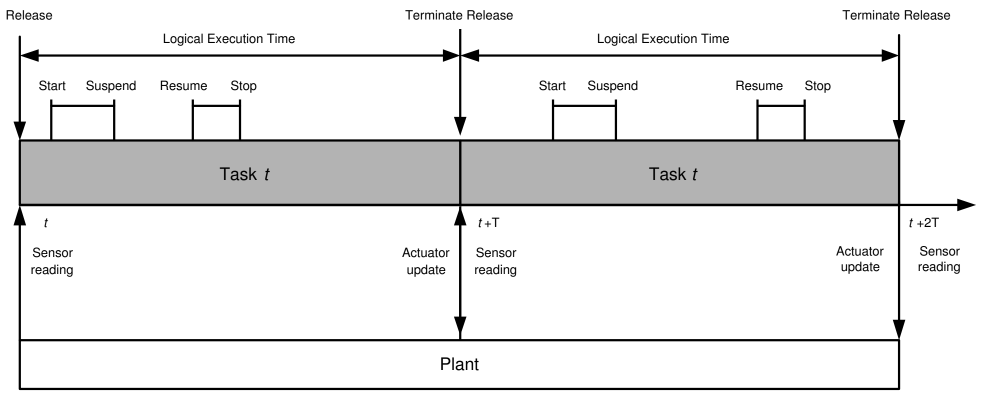

Figure 3. Execution of a task in time-triggered processing systems.

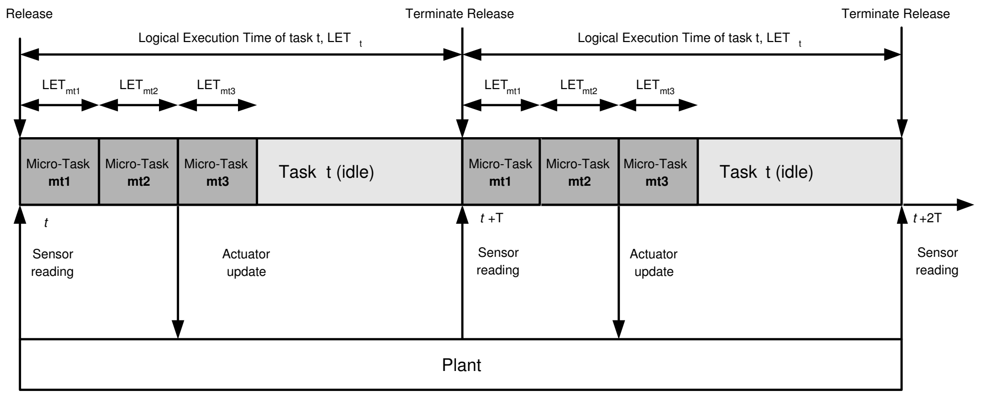

Figure 4. Execution of a task in time-triggered processing systems with micro-tasks.

end of the LET. When task `t` reads the sensor value at time t, it reacts on the plant only at time t+T. It logically reads again the sensor value at time t+T, i.e., immediately after it has updated the actuator. It computes the estimation and control law on the basis of the new sensor value. Then, it updates the actuator value at time t+2T. This means that it senses only at time t+2T the impact of the control law output, based on the sensor input at time t, i.e., two sampling periods later.

Figure 4 shows the same scenario and how the logical execution would be described using micro-tasks. The task `t` is split into three micro-tasks. Micro-task `m3` computes the part of the estimation that can be calculated before reading the sensor value, whereas micro-task `mt1` calculates the final state estimation using the current sensor value. Micro-task `mt2` computes the control law. Task `t` has LET_t and each micro-task has assigned its own LET, LET_{mt1} for micro-task `mt1`, LET_{mt2} for micro-task `mt2`, and LET_{m3} for micro-task `m3`, respectively. Actuator inputs can be made available at the end of the LET of the computation control law at LET_{mt2} and before the LET_t of task `t` has expired. Consequently, the impact of the sensor reading at time *t* can be sensed at time t+T. The advantage is an improvement of the systems reactivity.

5.4 Case Study Details

In the case study, we use task sequencing for error detection. The concept was outlined in Section 2. We employ the same structure of micro-tasks as it was applied in the estimator-based control system. To compute the expected output value of the plant we use a known model and check the computed value exp_y against the actual plant output y, reported by a sensor. If the difference between the values exceeds a certain threshold, we assume that a failure has been detected and an error has occurred.

Listing 4. TDL program with micro-tasks for error detection.

```
task error_detection [5 ms] {
  input double exp_y;
  input double y;
  output int alarm;
  uses perform_check (exp_y, y, alarm);
}

task plant_controller [5 ms] {
  input double y;
  input double r;
  output double u;
  uses compute_control (y, r, u);
}

task plant_model [20 ms] {
  input double u;
  output double exp_y;
  uses compute_model (u, exp_y);
}

task ctl_w_sanity {
  input double r, y;
  output int alarm;
  state double st_exp_y := 0;

  [LET = 10 ms]
  error_detection {y := this.y;
    exp_y := st_exp_y, alarm};

  [LET = 10 ms]
  if not_alarm(alarm) then
    plant_controller {y := this.y;
       r := this.r};

  if not_alarm(alarm) then
    actuator u := plant_controller.u;

  [LET = 30 ms]
  if not_alarm(alarm) then
    plant_model {u := plant_controller.u};
}

start mode main [100 ms] {
  task
    [1] ctl_w_sanity {r := s1; y := s2};
    ....
}
```

Listing 4 shows the TDL module fragment that implements the behavior of the case study. It specifies three micro-tasks: `error_detection`, `plant_controller`, and `plant_model`. These three micro-tasks are declared in their parent task `ctl_w_sanity`. Mapping the set of micro-tasks of Figure 4 to the task of Listing 4, micro-task `error_detection` corresponds to micro-task `mt1`, micro-task `plant_controller` to micro-task `mt2`, and micro-task `plant_model` to micro-task `m3`.

Listing 5. The E code generated from Listing 4.

```
  lbl1:    call, d[ctl_w_sanity]
2          call, d[error_detection]
           schedule, error_detection
4          future, 10, lbl2
           return
6 lbl2:    ifn, not_alarm, lbl3
           call, d[plant_controller]
8          schedule, plant_controller
  lbl3:    future, 10, lbl4
10         return
  lbl4:    ifn, not_alarm, lbl5
12         call, d[u]
           call, d[plant_model]
14         schedule, plant_model
  lbl5:    future, 80, lbl1
16         return
```

Listing 5 shows the E code resulting from the case study. The mode period starts at `lbl1` (Line 1). The driver of task `ctl_w_sanity` is executed, which updates the input ports `r` and `y`. The E machine executes the sanity check (Lines 2 to 5) by releasing the micro-task `error_detection`. The driver of the micro-task copies the value of the sensor reading to input port `y` and the calculated output value of task `plant_model` (stored in the state variable `st_exp_y`) to input port `exp_y`. As the LET of task `error_detection` equals 10 ms, the E machine passes control to the scheduler and resume after 10 ms (Line 4). At Label `lbl2`, the E machine checks, if an alarm has been set (by the micro-task `error_detection`). If the alarm has been set, the E machine immediately jumps to Label `lbl3`, waiting for 10 ms, and then continues at Label `lbl4`. This behavior is repeated at the beginning of each task-release sequence (Lines 6 and 11). In Line 6, if no alarm has been set, the E machine releases the micro-task `plant_controller` (Lines 7 to 10) and eventually reaches Label `lbl4` (Line 11). In this section (between Label `lbl4` and Label `lbl5`), the E machine calls the driver of actuator u (updating the actuator value) and releases the micro-task `plant_model`, which calculates the expected value exp_y of the next sensor reading. Finally at Label `lbl5`, the E machine returns control to the scheduler and resumes its operation from the beginning of the E code at Label `lbl1`.

6 Conclusion

The timing definition language (TDL) implements a timed computation model for hard real-time control systems. The timed computation model favors system determinism and predictability over code efficiency and system reactivity.

In this paper, we introduced the notion of micro-tasks and task sequencing to the programming model of TDL that prior to this work only supported periodic precedence-free execution of tasks. Micro-tasks and task sequencing are useful in reducing timing delays between sensor readings and actuator updates (e.g., for estimator-based control systems), managing startup and shutdown phases of control systems, and providing mechanisms for error-detection in fault tolerant systems (as presented in the case-study).

In our future work, we will extend the concept of micro-tasks to support the sequential execution of arbitrary patterns of micro-tasks, such as micro-task `mt1` and micro-task `mt2` are executed concurrently after micro-task `m0` has terminated.

7 Acknowledgments

We thank Klemens Winkler for his help in porting the E machine from RTLinux to RTEMS and the team of the Software Research Lab of the University of Salzburg for the lively and long discussions on the micro-task topic.

References

[1] G. Berry, 2000, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel. MIT Press .

[2] N. Halbwachs, 1997, *Synchronous Programming of Reactive Systems*. Kluwer.

[3] E. Coste-Maniere and N. Turro, 1997, The MAESTRO language and its environment: Specification, validation and control of robotic missions. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS97*, volume 2.

[4] J. Templ, 2004, TDL Specification and Report. Technical Report, Computer Science, University of Salzburg.

[5] B. Horowitz T. A. Henzinger, C. Kirsch, 2001, Giotto: A time-triggered language for embedded programming. In C. Kirsch, editor, *Proceedings of EMSOFT 2001*, volume 221 of *LNCS*. Springer.

[6] B. Horowitz, J. Liebman, C. Ma, T. John Koo, A. Sangiovanni-Vincentelli, and S. Sastry, 2003, Platform-Based Embedded Software Design and System Integration for Autonomous Vehicles. *IEEE Transactions*, 91(1):100 – 111.

[7] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree, 2002, A Giotto-based helicopter control system. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 46–60. Springer.

[8] B. Horowitz T.A. Henzinger and C.M. Kirsch, 2003 *Software-Enabled Control: Information Technology for Dynamical Systems*, pages 123–146. IEEE Press and Wiley-Interscience.

[9] Christoph M. Kirsch, 2002, Principles of Real-Time Programming. *LNCS*, 2491.

[10] A. Ghosal, T.A. Henzinger, C.M. Kirsch, and M.A.A. Sanvido, 2004, Event-driven programming with logical execution times. In

International Workshop on Hybrid Systems: Computational and Control, number 2993 in Springer LNCS, pages 357 – 371.

[11] Gene F. Franklin, David J. Powell, and Michael L. Workman, 1997, *Digital Control of Dynamic Systems*. Prentice Hall.

[12] H. Kopetz and R. Nosssal, 1997, Temporal Firewalls in Large Distributed Real-Time Systems. In *Proceedings of IEEE Workshop on Future Trends in Distributed Computing*.

[13] H. Kopetz, 1998, The Time-Triggered Model of Computation. *Proceedings of the 19th IEEE Systems Symposium (RTSS98), December 1998*.

[14] N.G. Leveson, 1995, *Safeware System, Safety and Computers*. Addison Wesley.

[15] Windriver web site, 2004, `http://www.windriver.com/`.

[16] *OSEK/VDX Operating System Sepcification 2.2*, 2001. Version 2.2.

[17] G. Berry, 1999, *The Constructive Semantics of Esterel*. Number 3.0.

[18] P. Caspi, C. Mazuet, R. Salem, and D. Weber, 1999, Formal Design of Distributed Control Systems with Lustre. In M. Felici, K. Kanoun, and A. Pasquini, editors, *SAFECOMP99*, Springer LNCS 1698, pages 396 – 409.

[19] S. Vestal. *MetaH Users Manual*. Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, version 1.27 edition.

[20] W. Pree and J. Templ, 2004, On Digital Controllers and the Giotto Programming Model. Technical report, Department of Computer Science, University of Salzburg, Austria.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, 1995, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

[22] C. Kirsch T. A. Henzinger, 2002, The embedded machine: Predictable, portable real-time code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[23] RTEMS official web site, 2004, `http://www.rtems.com`.

[24] OAR Corporation, 2004, *RTEMS On-Line Library*, 4.6.1 edition.

[25] GNU compiler collection - official web site, 2004, `http://gcc.gnu.org/`.

[26] GNU debugger - official web site, 2004, `http://www.gnu.org/software/gdb/gdb.html`.