Migration of Legacy Software Towards Correct-by-Construction Timing Behavior

Stefan Resmerita[1], Kenneth Butts[2], Patricia Derler[1], Andreas Naderlinger[1], Wolfgang Pree[1]

[1]C. Doppler Laboratory Embedded Software Systems, Univ. Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

[2]Toyota Technical Center
1555 Woodridge, Ann Arbor, Michigan 48105
ken.butts@tema.toyota.com

Abstract. This paper presents an approach for incrementally adjusting the timing behavior of legacy real-time software according to explicit timing specifications expressed in the Timing Definition Language (TDL). The primary goal of such a migration is to achieve predictable timing behavior, which enables application of formal verification methods to the legacy system.
Our approach entails a minimal instrumentation of the original code combined with an automatically generated runtime system, which ensures that the executions of designated periodic computations in the legacy software satisfy the logical execution time specifications of the TDL model. The presented approach has been applied to a complex legacy controller system in the automotive domain.

1 Introduction

Modern methodologies for embedded system design such as Model-Driven Engineering (MDE) [1] and Platform-Based Design [2] advocate a top-down approach for application development. The development process starts from high level models, which are incrementally refined to software models and then to implementations on execution platforms.

While the benefits of these approaches are well-understood, their full adoption in the established embedded industry is rather slow. One of the main factors responsible for this is the large base of legacy applications, which have been traditionally developed at the programming language level, are usually highly optimized and thoroughly tested. MDE is thus employed only partially, typically for developing new functionality up to the software model, which is then manually merged with the existing legacy code.

Based on the argument that execution time of software should be captured in high-level models [3], several MDE approaches propose programming disciplines based on timing models for certain embedded applications. In particular,

the Logical Execution Time (LET) abstraction has been proposed for achieving predictable timing properties of control applications [4]. This model is used in several timing specification languages and tools such as the Hierarchical Timing Language (HTL) [5] and the Timing Definition Language (TDL) [6]. While all of these assume the classical MDE top-down approach, they are particularly amenable to a bottom-up application to legacy software, due to the separation of concerns provided by the original LET model, where timing is separated from functionality. This facilitates the enforcement of timing requirements on a legacy system in a systematic and minimally interventive way. It also addresses intellectual property concerns, requiring no information about what the legacy code does. However, availability of the legacy source code and platform configuration information is assumed.

In this paper we describe how to apply TDL modeling to typical controller systems. We propose an instrumentation-based approach, with minimal intervention in the legacy code and platform configuration. To achieve this, we had to reconcile the top-down approach of TDL with the constraints imposed by the legacy system. Two main aspects required trade-offs in this respect: (1) Event-triggered computations, which in TDL are assumed to have lower priorities than time-triggered tasks, while the legacy application has higher priority events, and (2) the TDL runtime system, which originally implements a virtual machine called E-Machine and compiles the timing specification into code for this E-Machine, called E-code, which has proved to be quite large for complex legacy applications. Issue (1) was addressed by a careful scheduling analysis, considering information about minimum inter-arrival times of high-priority events. Problem (2) was resolved by employing an application-specific runtime system, called TDL-Machine, which was code-generated from the TDL model and from application-specific information.

The approach described in this paper was applied to a complex industrial legacy application in an incremental manner. The desired timing behavior was tested by software-in-the-loop and hardware-in-the-loop simulations.

2 Background

This section briefly presents the Timing Definition Language (TDL), as well as common characteristics of legacy software that challenge some of the assumptions made in LET-based programming disciplines such as TDL.

2.1 The Timing Definition Language (TDL)

TDL allows the LET-based specification of timing properties of hard real-time applications. The LET of a computational unit, or task, represents a fixed logical duration between the time instant when the task becomes ready for execution and the instant when the execution finishes. A task's LET is specified at the model level, independently of the task's functionality. When deploying the model on a platform, the LET specification is satisfied if the total physical execution

time of the task is within the LET interval for every task invocation, and an appropriate runtime system ensures that task inputs are read at the beginning of the LET interval (the release time) and task outputs are made available at the end of the LET interval (the termination time). This is illustrated in Figure 1. Between release and termination points, the output values are those calculated in the previous execution. Default or specified initial values are used in the first execution of a task.

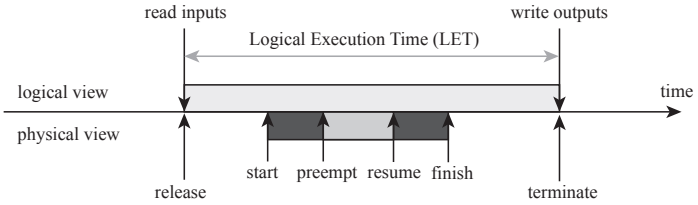

Fig. 1. The Logical Execution Time (LET)

Tasks can receive information from the environment via sensors and act on the environment via actuators. A task has input ports, output ports, and state ports. State ports keep state information between different executions of the same task.

Tasks that are executed concurrently are grouped in modes. In TDL, a mode is a set of periodically executed activities: task invocations, actuator updates, and mode switches. Such a mode activity has a specified execution rate and may be carried out conditionally. The LET of a task is expressed as the mode period divided by the frequency of the task invocation. Note that the time steps of all activities in a mode period can be statically determined.

Mode activities are carried out by a runtime system which performs the following operations at every time step:

- Update output ports of tasks whose LET end at the current time step. At time 0, the ports are initialized rather than updated.
- Update actuators.
- Test for mode switches. If a mode switch is enabled, switch to the target mode.
- Update input ports of the tasks whose LET start at the current time step.
- Trigger the execution of the tasks whose LET start at the current time step.

TDL provides a top level structuring unit called a module, which groups sensors, actuators, tasks, and modes that belong together. The module concept serves multiple purposes: (1) a module provides a name space and an export/import mechanism and thereby supports decomposition of large systems, (2) modules allow the parallel composition of real-time applications, (3) modules serve as units of loading, that is, a runtime system may support dynamic loading and

```
1  module Sender {
2      sensor int s1 uses getS1;
3      actuator int a1 uses setA1;
4      public task inc {
5          input int i;
6          output int o := 10;
7          uses incImpl(i,o);
8      }
9      start mode main [period=5ms] {
10         task [freq=1] inc(s1); //LET = 5ms (=period/freq)
11         actuator [freq=1] a1 := inc.o;
12         mode [freq=1] if exitMain(s1) then freeze;
13     }
14     mode freeze [period=1000ms] {}
15 }
```

Listing 1.1. A TDL example

unloading of modules, and (4) modules are the natural choice as unit of distribution because dataflow within a module (cohesion) will most probably be much larger than dataflow across module boundaries (adhesion).

An example of a TDL program is shown in Listing 1.1 and a graphical representation of this program is shown in Figure 2. In the example, module Sender contains a sensor variable $s1$ and an actuator variable $a1$. The value of $s1$ is updated by executing the (platformspecific) driver $getS1$ and the value of $a1$ is sent to the physical actuator by using the platform-specific driver $setA1$. Every module has exactly one start mode, indicated by preceding the mode declaration with the keyword start. The declaration of the output port of task inc specifies an initial value of 10. The task is invoked in mode main of the Sender module, where its input port is connected to the sensor $s1$. In the same mode, actuator $a1$ is updated with the value of the task's output port. The timing behavior of the mode activities is specified by means of individual frequencies within their common mode period. For example, with a frequency of 1, task inc is defined to have a LET of 5 ms. A more detailed description of TDL features can be found in [7].

A TDL application consists of a set of time-triggered tasks and a runtime system called TDL-Machine, which performs all mode operations according to the TDL specifications. A platform specific implementation of the TDL-Machine can be generated from the specifications [8].

2.2 Aspects of Legacy Controller Systems

It is common for an embedded controller software system to contain both time-triggered and event-triggered computations. Some event-triggered tasks may require fast reaction times, and thus may have higher priorities than time-triggered tasks.

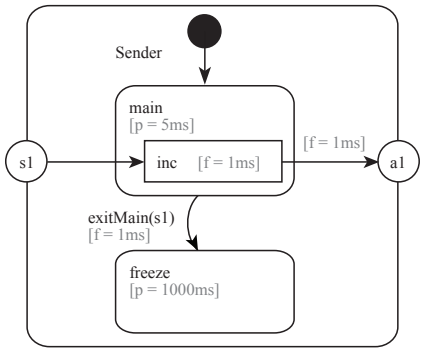

Fig. 2. A TDL example

Legacy operating systems require tasks to be split into execution units, also called tasks. We refer to such a task as a *platform task*. Typically the number of tasks is restricted. For example, OSEK/VDX [9] or AUTOSAR OS [10] suggest a maximum of 8 to 16 tasks. Complex systems often comprise more tasks thus a common design practice is to group the time-triggered computations into a small number of time-triggered platform tasks, which are triggered by a high priority task (also called sequencer or dispatcher task [11]) that is itself triggered from a periodic interrupt that defines the base period in the system. That high priority task dispatches the time-triggered platform tasks at multiples of the base period, using system services for task activation. In addition, each time-triggered task may internally perform computations at multiples of the task's period.

Another common characteristic of legacy embedded code is heavy usage of shared memory communication (global variables) between various components in the system. Moreover, communication with the physical environment is done by memory-mapped I/O devices. Thus, reading from a sensor means accessing a (read-only) global variable, while actuating means writing into a global variable.

3 Modeling Timing Behavior of Legacy Controllers with TDL

The main goal of imposing a LET-based execution and data-transfer semantics on an existing application is to eliminate unpredictable behaviors due to variations in execution times. An important refactoring requirement in this respect is minimal modification of the legacy system, including the application code and its configuration on the platform. Thus, code changes are done only by adding the TDL-Machine as a separate component and by inserting calls to the TDL-Machine functions at well defined, top-level places in the original application. No line of the legacy code is modified. Also, all the parameters of the legacy configuration remain unchanged (same platform tasks, periods, and priorities). Additional resources of the operating system may be necessary to trigger execu-

tions of TDL tasks, as described in the sequel. Moreover, an additional platform task may be required for the TDL-Machine.

Modeling the timing behavior of legacy software with TDL must reconcile the assumptions made on the implementation of TDL tasks and the ability of the runtime system to control executions with the characteristics of the legacy applications mentioned in Section 2.2. TDL requires that inputs and outputs of TDL tasks be passed to the implementation functions by means of function arguments, while the legacy code uses mostly global variables. Platform tasks are activated according to the legacy configuration, which must remain unchanged, so the TDL runtime system does not have full control over triggering of TDL tasks. Nevertheless, one has to make sure that the TDL semantics is preserved.

Complex legacy applications contain periodic computations with periods that differ by several orders of magnitude. For example, a computation may have a period of 5 milliseconds, while another one may have a period of 3 seconds. Since each periodic computation is mapped to a TDL task, the number of operations of the TDL-Machine in a hyperperiod of the system (the least common multiple of all the periods) may be quite large. This makes the usage of the E-Machine approach originally proposed in Giotto [4] and later used in [12] and [13] impractical due to memory constraints, since the E-code defines all operations in a hyperperiod. Thus, we chose to generate directly from the TDL specification a TDL-Machine specific to the particular legacy system, rather than generating E-code and using a generic implementation of an E-Machine.

3.1 Mapping the legacy architecture to TDL constructs

A TDL task can be mapped to any function of the legacy code, which is referred to as the implementation function of the task. Since TDL tasks are assumed to be independent, a TDL task cannot be mapped to a function that is called from the implementation function of another TDL task.

In general, TDL tasks are included in platform tasks, in the sense that a platform task may contain implementation functions of more than one TDL task, while a TDL task cannot be mapped to more than one platform task. If a TDL task is mapped to a platform task, the platform task function is the implementation function of the TDL task. A TDL task can be:

- Synchronous, also called time-triggered, if it corresponds to a periodic computation and it has a LET specification. The period of a synchronous TDL task is the same as the period of execution of the implementation function of the TDL task, which is a multiple of the period of execution of the platform task that contains the implementation function. The LET intervals are established together with application engineers, such that the system is schedulable. TDL has been extended to allow the definition a task's LET inside the task's period, by specifying an offset between the beginning of the period and the beginning of the LET.
- Asynchronous, also called event-triggered, if it corresponds to an event-triggered computation, in which case it has no LET. The task implementa-

tion function of an asynchronous TDL task always corresponds to a platform task function.

An input (output) port of a TDL task T corresponds to a legacy global variable that is read (written) during the execution of the task implementation function and that is written (read) in another part of the legacy application that is independent of the particular TDL task T.

We consider the typical case of memory-mapped I/O devices, where sensors and actuator values are stored in memory locations (global variables mapped to hardware registers). Thus, sensing is performed by first writing in an output variable (which, for example, can be mapped to a command register of an A/D converter) and then reading from an input variable (which, for example, can be mapped to the data register of the A/D converter). Consequently, the TDL model contains no dedicated sensor/actuator variables.

Since we deal here with the migration of monolithic legacy controllers, we define one TDL module per application. A TDL module may contain several modes. We consider here the case where all TDL tasks are present in each mode, such that modes only define different timing behaviors of the tasks.

3.2 Implementation of the TDL operational semantics

TDL operations are carried out at runtime by a dedicated component called the TDL-Machine, which deals with activation of synchronous TDL tasks, data transfer, and mode switches. The architecture of the TDL-Machine and its interaction with the legacy application are schematically illustrated in Figure **??**. The TDL-Machine has a time-triggered component and an event-triggered component.

The time-triggered component is executed in a periodic platform task with the highest priority and smallest period (the base period). Such a task is common in legacy applications, its main role being to dispatch executions of lower-priority periodic tasks with periods that are multiples of the base period. If the task is not defined in the legacy application, or if the TDL-Machine needs a smaller base period (e.g., for a finer granularity of LET endpoints), then an additional platform task needs to be introduced. The time-triggered component performs all the operations that are necessary at LET endpoints. The operations that interfere with the execution of legacy code are synchronous task invocations and data transfers. We describe how to deal with these situations in the sections below. Mode switches are implemented by simply changing the LETs for the task set. These LET intervals for each mode are stored in a table.

The event-triggered component defines one start function and one end function for each TDL task. These functions are called whenever an execution of the task implementation (legacy) function begins, respectively ends. Thus, calls to these functions are inserted at the beginning and end of the corresponding legacy functions. Their role is to perform buffering and synchronization operations, as detailed further in this section.

Listing 1.2 sketches the implementation of a LET-based task.

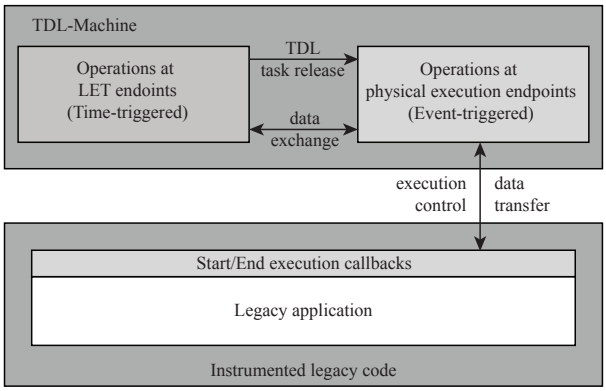

Fig. 3. The TDL-Machine architecture

```
1 void Tx_Implementation() {
2     ...                              // local variables
3     On_Tx_start_execution();         // execution start callback
4     ...                              // legacy code
5     On_Tx_end_execution();           // execution end callback
6 }
```

Listing 1.2. LET-based operational semantics

3.3 Activation of Synchronous TDL Tasks

In every execution period, a time-triggered TDL task must be activated at the start of its LET interval. The execution period is a multiple of the period of the platform task that contains the TDL task implementation function. If the start of the LET interval coincides with the start of the period, then no action is taken by the TDL-Machine, since every platform task is activated by the platform anyway. If the LET interval starts after the beginning of the TDL task's period (at a fixed offset), then an additional synchronization point is needed, to enable the TDL-Machine to trigger the execution of the task implementation function at the LET start, which ensures that the physical execution of the function takes place within the LET bounds. The implementation of this synchronization is operating system-specific. For example, in the case of an OSEK operating system, a WaitEvent system call is used in the entry instrumentation function. At runtime, a corresponding SetEvent system call is performed by the TDL-Machine at the LET start. This ensures that the legacy function does not start executing before the LET start. No change is made to the platform-triggered activation of platform tasks (which include the asynchronous TDL tasks).

For example, consider a platform task with a period of 5ms, with a task function as described in Listing 1.3.

```
1  void Platform_Task() {
2      taskCounter = taskCounter + 1;
3      legacy_func_5ms();              // executed every 5ms
4      if (taskCounter & 0x01) {
5          legacy_func_10ms();         // executed every 10ms
6      }
7  }
```

Listing 1.3. Platform task

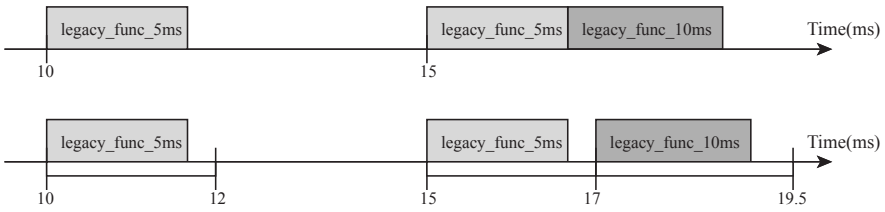

Fig. 4. Execution example: original (top) and with LETs (bottom)

We define a TDL task T5 with a period of 5ms and implementation function `legacy_func_5ms`, and a TDL task T10 with a period of 10ms and implementation function `legacy_func_10ms`. Assume that T5 has a LET of 2ms and offset zero, while T10 has a LET of 2.5ms and an offset of 2ms. An example of executions in the original system and in the TDL-modeled system is depicted in Figure 4.

In the TDL-based system, callbacks are inserted at the beginning and end of the two legacy functions. To be able to enforce a LET start for the 10ms function, the callback `On_T_10ms_start_execution` makes a blocking call on an operating system resource, which is released at the LET start by the time-triggered component of the TDL-Machine as described in Listing 1.4.

3.4 Data Transfer Operations

The implementation of LET-based data transfer for a synchronous TDL task must deal with the fact that data communication between the legacy implementation function of the TDL task and the rest of the system is done by shared memory. Since inputs and outputs are provided via global variables, their values need to be buffered in TDL-specific variables as described below. The following behavior must be ensured:

(A) When the LET begins, the value of each original input variable is stored in an additional internal task input variable. This is necessary because the value of an original input variable may change between the starting of the LET and

```
1  // called at the beginning of legacy_func_10ms
2  void On_T_10ms_start_execution() {
3      WaitEvent(EV_T10_LET_START);
4  }
5  // time-triggered LET scheduler function
6  void LETSchedulerStep(double time) {
7      ...
8      if (time == LET_START_T10ms){
9          SetEvent(Platform_Task, EV_T10_LET_START);
10     }
11     ...
12 }
```

Listing 1.4. Example implementation of 2 time-triggered tasks

the moment when the physical execution of the TDL task implementation function starts.

(B) During execution, the task implementation function uses the values of the internal input variables instead of the actual values of the original variables.

(C) During execution, the values of legacy output variables are stored in additional internal output

(D) When LET ends, the original global output variables (the legacy variables) are updated with the values of the TDL-internal output variables.

Operations A and D are executed by the time-triggered part of the TDL-Machine. To achieve a minimal instrumentation of the legacy code, we chose to implement B and C in the execution callbacks, as explained below.

Communication Between Synchronous TDL Tasks Only Consider two periodic time-triggered legacy functions `tt_func_write` and `tt_func_read`, where some execution of `tt_func_write` updates a global variable `gvar`, and some execution of `tt_func_read` reads from the same variable. Assume now that `tt_func_write` is mapped to a TDL task `T_WRITE` and `tt_func_read` is mapped to another TDL task `T_READ`. A sample module with these two TDL tasks is described in Listing 1.5.

To ensure LET-based data transfer between the functions `tt_func_write` and `tt_func_read`, the TDL-Machine is generated so that it has an internal output variable for `T_WRITE` called `T_WRITE_tp_o_gvar`, a task output port variable called `T_WRITE_o_gvar`, and an input port variable for `T_READ`, called `T_READ_in_gvar`. The TDL-Machine also uses additional buffer variables `T_WRITE_tp_gvar` and `T_READ_tp_gvar`. The following data transfer callbacks are defined in the TDL-Machine:

- `On_T_WRITE_start_execution` and `On_T_WRITE_end_execution`,
- `On_T_READ_start_execution` and `On_T_READ_end_execution`.

```
1  module Example {
2      public task T_WRITE {
3          output int gvar := 0;
4          uses T_WRITE_Implementation(gvar);
5      }
6      public task T_READ {
7          input int gvar;
8          uses T_READ_Implementation(gvar);
9      }
10     start mode main [period=5ms] {
11         task [freq=1] T_WRITE();     //LET = 5ms, offset = 0ms
12         task [freq=5, slots=2-4]
13                  T_READ(T_WRITE.o); //LET = 3ms, offset = 1ms
14     }
15 }
```

Listing 1.5. TDL module with 2 tasks

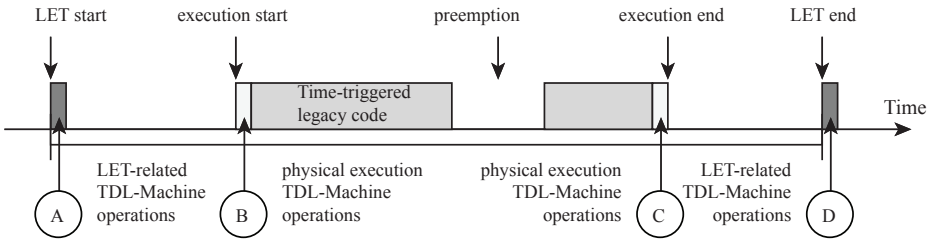

Fig. 5. Data transfer operations for LET-based tasks

Figure 5 shows a sample execution trace which highlights when data transfer operations of the TDL-Machine are executed. The operations at LET endpoints A and D are described in Listing 1.6 and operations at physical execution endpoints B and C are described in Listing 1.7.

Communication Between Synchronous and Asynchronous TDL Tasks Consider three legacy functions `tt_read_write`, `ev_write` and `ev_read`, with the corresponding synchronous TDL task `T_RW`, and the two asynchronous TDL tasks `E_WRITE`, and `E_READ`, respectively. Assume that `tt_read_write` reads variable `gvar_r` and writes into variable `gvar_w`, `ev_write` writes in variable `gvar_r`, and `ev_read` reads from both variables. An execution example is shown in Figure 6, and the corresponding TDL-Machine operations are summarized in Listings 1.8, 1.9 and 1.10.

For this example, one can check that the LET requirements for data transfer regarding synchronous TDL tasks are satisfied. In particular, the value of `gvar_r` read in the execution of `tt_read_write` is the one updated by a previous execution of `ev_write`, which is not shown in the figure (the one preceding

```
1  void TDLMachineStep(time) {
2      if (time == LET_START_T_READ) {                // A
3          T_READ_in_gvar = T_WRITE_o_gvar;
4      }
5      if (time == LET_END_T_WRITE) {                 // D
6          T_WRITE_o_gvar = T_WRITE_tp_o_gvar;
7      }
8  }
```

Listing 1.6. TDL-Machine step

```
1  void On_T_WRITE_start_execution() {                // B
2    T_WRITE_tp_gvar = gvar;
3  }
4  void On_T_WRITE_end_execution() {                  // C
5    T_WRITE_tp_o_gvar = gvar;
6    gvar = T_WRITE_tp_gvar;
7  }
8  void On_T_READ_start_execution() {                 // B
9    T_READ_tp_gvar = gvar;
10   gvar = T_READ_in_gvar;
11 }
12 void On_T_READ_end_execution() {                   // C
13   gvar = T_READ_tp_gvar;
14 }
```

Listing 1.7. Operations at physical endpoints

```
module Example {
    public task T_RW {
        input int gvar_r;
        output int gvar_w := 0;
        uses T_RW_Implementation(gvar_r, gvar_w);
    }
    public task E_WRITE {
        output int gvar_r :=0;
        uses E_WRITE_Implementation(gvar_r);
    }
    public task E_READ {
        input int gvar_r;
        input int gvar_w;
        uses E_READ_Implementation(gvar_r, gvar_w);
    }
    start mode main [period=5ms] {
        task [freq=1] T_RW(E_WRITE.gvar_r);
    }
    asynchronous {
        E_WRITE();
        E_READ(E_WRITE.gvar_r, T_RW.gvar_w);
    }
}
```

Listing 1.8. TDL program

```
void TDLMachineStep(time) {
    if (time == LET_START_T_RW) {                    // A
        T_RW_in_gvar_r = E_WRITE_o_gvar_r;
    }
    if (time == LET_END_T_RW) {                      // D
        T_RW_o_gvar_w = T_RW_tp_o_gvar_w;
    }
}
```

Listing 1.9. Operations at LET endpoints

```
1  void On_E_WRITE_start_execution() {                    // W
2      E_WRITE_tp_gvar_r = gvar_r;
3  }
4  void On_E_WRITE_end_execution() {                      // X
5      E_WRITE_o_gvar_r = gvar_r;
6      gvar_r = E_WRITE_tp_gvar_r;
7  }
8  void On_E_READ_start_execution() {                     // Y
9      E_READ_tp_gvar_r = gvar_r;
10     E_READ_tp_gvar_w = gvar_w;
11     gvar_r = E_WRITE_o_gvar_r;
12     gvar_w = T_RW_o_gvar_w;
13 }
14 void On_E_READ_end_execution() {                       // Z
15     gvar_w = E_READ_tp_gvar_w;
16     gvar_r = E_READ_tp_gvar_r;
17 }
18 void On_T_RW_start_execution() {                       // B
19     T_RW_tp_gvar_r = gvar_r;
20     T_RW_tp_gvar_w = gvar_w;
21     gvar_r = E_WRITE_o_gvar_r;
22 }
23 void On_T_RW_end_execution() {                         // C
24     T_RW_tp_o_gvar_w = gvar_w;
25     gvar_r = T_RW_tp_gvar_r;
26     gvar_w = T_RW_tp_gvar_w;
27 }
```

Listing 1.10. Operations at physical execution endpoints

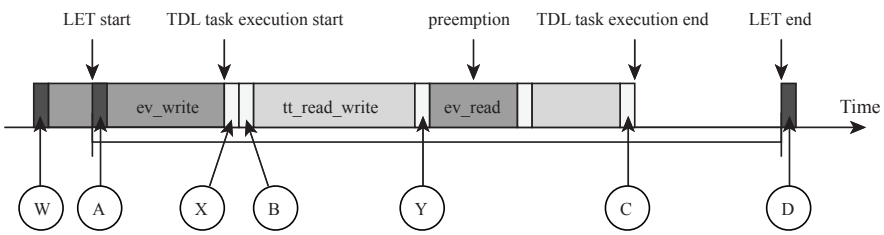

Fig. 6. Mixed time and event-triggered execution example

the depicted execution). This is the value of the output port of `E_WRITE` at the beginning of `T_RW`'s LET. However, `ev_read` uses the latest value of `gvar_r`, updated during the depicted execution of `ev_write`. Thus, TDL modeling may introduce controlled delays in the communication involving synchronous TDL tasks, but it never delays communication between asynchronous tasks.

4 Industrial application

The approach presented in this paper has been applied to an industrial engine control software, which comprises millions of lines of code and runs on top of an operating system with fixed priority scheduling. From the three periodic platform tasks, we identified 12 TDL tasks with periods between 1 millisecond and 2 seconds. The TDL tasks communicate via approximately 300 global variables. The application also has multiple event-triggered tasks, with the highest priority being given to the crank angle event. The TDL modeling procedure was optimized in order to decrease the additional memory required by TDL buffers, based on the following observations:

- If data transfer through a global variable cannot be affected by preemption, then no buffering is needed for that variable. For example, if all the readers and writers of a variable are part of the same platform task then the variable requires no buffering. Another example in this respect is a variable written only by an event-triggered task with lower priority than all the TDL reader tasks.
- Buffering can be reduced by directly substituting in the code original variables with TDL internal variables. For example, if a variable is written only by one TDL task, the variable can be substituted in the task's code with the corresponding TDL output buffer variable, eliminating the need to buffer the original variable during the physical execution of the task. This buffering is illustrated in Listing 1.7, where in our example the occurrences of `gvar` in the task's code are replaced by `T_WRITE_tp_o_gvar`, and `T_WRITE_tp_gvar` is eliminated. A similar reduction can be made for the cases where only one TDL task reads from an input variable.

4.1 Determining the Logical Execution Time

The two main design parameters related to the LET are the offset,i.e., the start of the LET in a task period, and the size of the LET. The LET must be at least as large as the worst case reaction time (WCRT) of the task, in order to ensure that the task is schedulable, i.e., every execution of the task ends before the end of the corresponding LET interval. Note that the maximum execution time of task T in the time interval $[t_1, t_2]$ is the maximum number of executions of T (in that interval) multiplied by the worst case execution time of T, ($WCET(T)$):

$$\left\lceil \frac{t_2 - t_1}{\pi_T} \right\rceil * WCET(T),$$

where π_T denotes the invocation period of T. If T is event-triggered, then it is considered periodic with period equal to the minimum inter-arrival time of the triggering event in the current operating mode.

Let the set $\mathbf{T}$ of all tasks $T_i \in \mathbf{T}$ be ordered according to their priorities, where $i = 1$ means highest priority. The worst case response time of task T_i is given by the smallest fixed point of the following recursive equation [14, 15]:

$$R(T_i)^{(k+1)} = WCET(T_i) + \sum_{j=1}^{(i-1)} \left\lceil \frac{R(T_i)^{(k)}}{\pi_{T_j}} \right\rceil * WCET(T_j)$$

with $R(T_i)^0 = WCET(T_i)$. If no fixed point exists, the task is not schedulable.

The LET size is a trade-off between opposite requirements. For example, robustness requires larger LETs, while control performance requires small LETs, to minimize the additional reaction time incurred due to the LET execution semantics. The trade-off is inherently dynamic, since the relative importance of one requirement or another depends on the operating conditions at a given time.

For the engine control application, we have chosen to specify multiple modes of timing behavior, spanning the entire range of engine speeds, as follows:

- At high engine speeds, when the computational load is severe and fast response times are crucial, the LET of a TDL task is equal to the worst case reaction time of the task.
- At low engine speeds, the LET of a task can be larger by up to 20%. This leaves room for adding and testing new functions without affecting the timing behavior of the application.

Several TDL modes have been defined between the highest and the lowest engine speeds. Each mode has the same tasks, the only difference between modes being in the tasks' LETs. The modes have been established based on WCRT profiles of the TDL task functions, which represent the minimum LET for each engine speed. TDL mode switches are triggered by the engine speed variable, which is an input to the TDL:Machine. Figure 7 shows an example of a WCRT profile which defines five possible timing modes. Note that the LET of task T

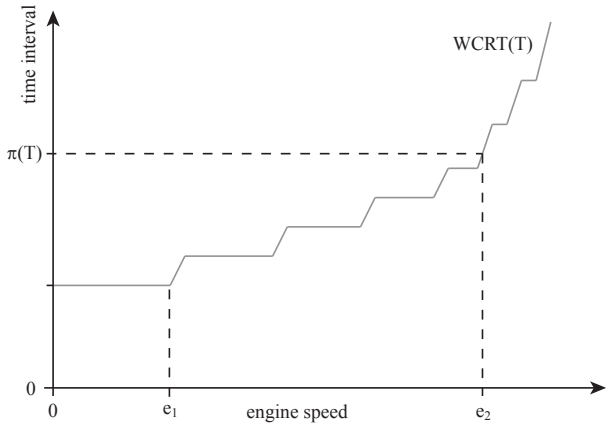

Fig. 7. Worst case execution time of task T as function of the engine speed

cannot exceed the task's period $\pi(T)$. Thus, at engine speeds higher than e_2 the task is deemed unschedulable.

The offset of a TDL task T was determined according to the "place" of the TDL task in the owner platform task. In principle, the LET start of a TDL task is the same as the LET end of the preceding TDL task in the same platform task. A TDL task which is always executed first in its platform task has an offset equal to zero. For example, in listing 1.3 and Figure 4 the task with period 5ms has offset zero (it is always executed first), while the offset of the 10ms task (always executed second) equals the LET of the first task.

The WCETs have been estimated by using the $a3$ tool [16]. This tool relies on an accurate model of the processor, which was not available at the time of testing. Therefore, a "vanilla" version of the processor was used and consequently the estimates were quite conservative.

4.2 Testing Results

The application modeled with TDL has been tested in a software-in-the-loop (SIL) simulator [17] [18], as well as in a hardware-in-the-loop testbed(HIL). The SIL testing compared signals from the original and TDL-based applications when both were running in parallel, in closed-loop with an engine model, as schematically shown in Figure 8. A sample of the testing results is provided in Figure 9, where one can observe that the behavior of the TDL-modeled system is close to the original one, modulo some small delays introduced by the LET behavior. The delays resulted from using the worst-case scenario for setting the LET of the task and from overestimation of WCET of tasks. They could be reduced by using more accurate execution time estimations.

The HIL testing was performed in a testbed where the original and TDL-based applications were executed on the same Electronic Control Unit interconnected with a real-time computer running a model of the engine. Signals were

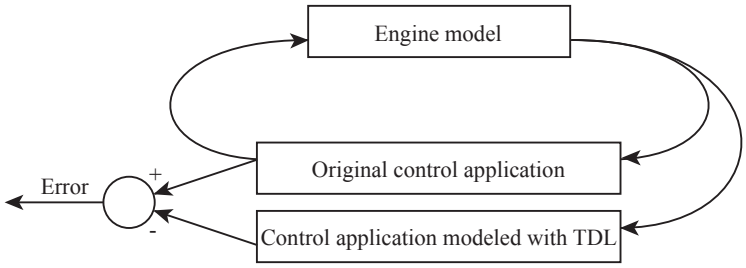

Fig. 8. SIL setup for comparing TDL-based application with original application

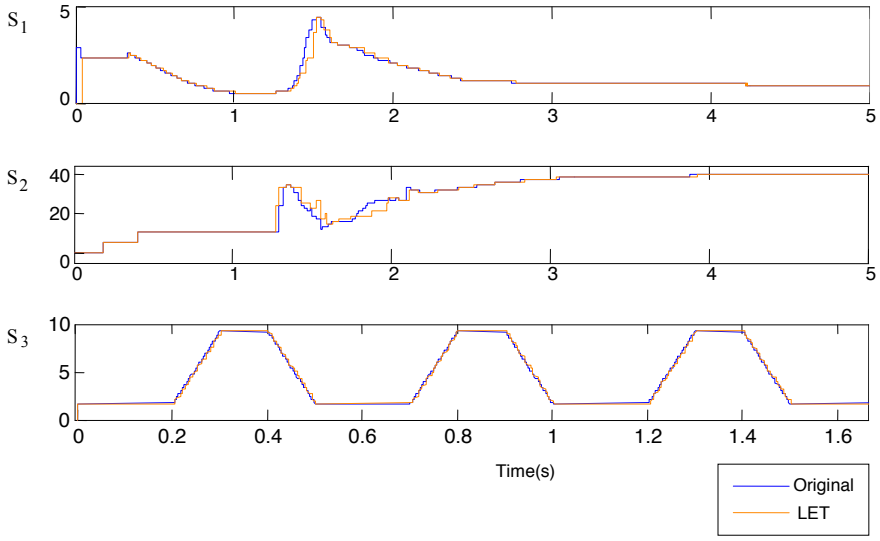

Fig. 9. Signal comparison

sampled with a period of 50 microseconds and then compared. In one of the test cases, the robustness of the system was tested by making a non-functional modification in the code and comparing the outputs. In the original software, the change led to a difference in the outputs, while no difference was exhibited in the TDL-modeled version.

Usually, a task function contains a sequence of calls to multiple process functions that need to be executed when the task is triggered. Many such process functions are independent and conceptually the order in which they are called does not matter. We have chosen a periodic task T and moved a process function f from the beginning of the task T to the end. The function f updates a global variable v_2 with the value of another global variable v_1. The latter is updated by an event-triggered task E, which has higher priority than T. The variables v_1 and v_2 are regarded as an input, respectively an output of T. Case A in Figure 10 illustrates an execution of T in the original application, where f is called at

the beginning of T's execution. Then, the value of v_2 is set to the value of v_1. Thereafter, E preempts the execution of T and updates v_1. Case B shows an execution of T in the modified application, where f is executed at the end of T's execution. Here, the variable v_2 is updated after E's preemptive execution, and consequently the output of T is different. Notice that no difference occurs when E does not preempt. This is an example where different ways of serializing executions of concurrent components lead to different behaviors in the system, due to preemption from event-triggered tasks. The TDL-modeled system is robust with respect to such changes, as demonstrated by the corresponding cases in Figure 11. In this case, the TDL task T was the only writer of v_2, hence in the TDL version the occurrence of v_2 in the code of f was replaced by a TDL buffer output variable $buffer\_v2$. Notice that this is updated with the value of v_1 when f is executed and then v_2 is updated with the value of $buffer\_v2$ at the end of the LET. One can see that the output of T does not change from case A to case B.

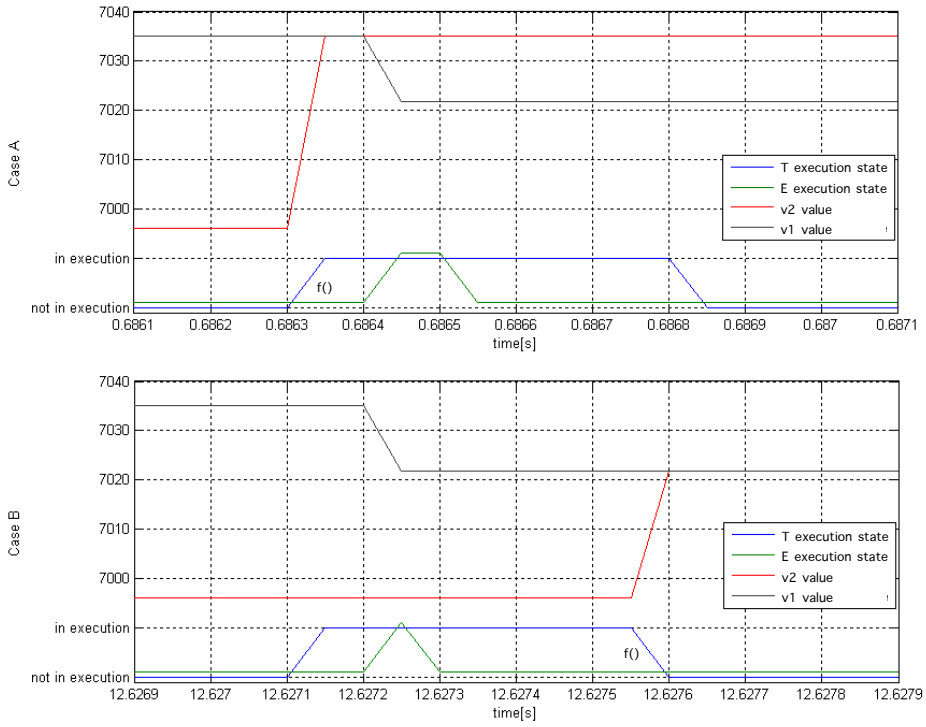

Fig. 10. Changing the place of a function call leads to different values of the output v_2

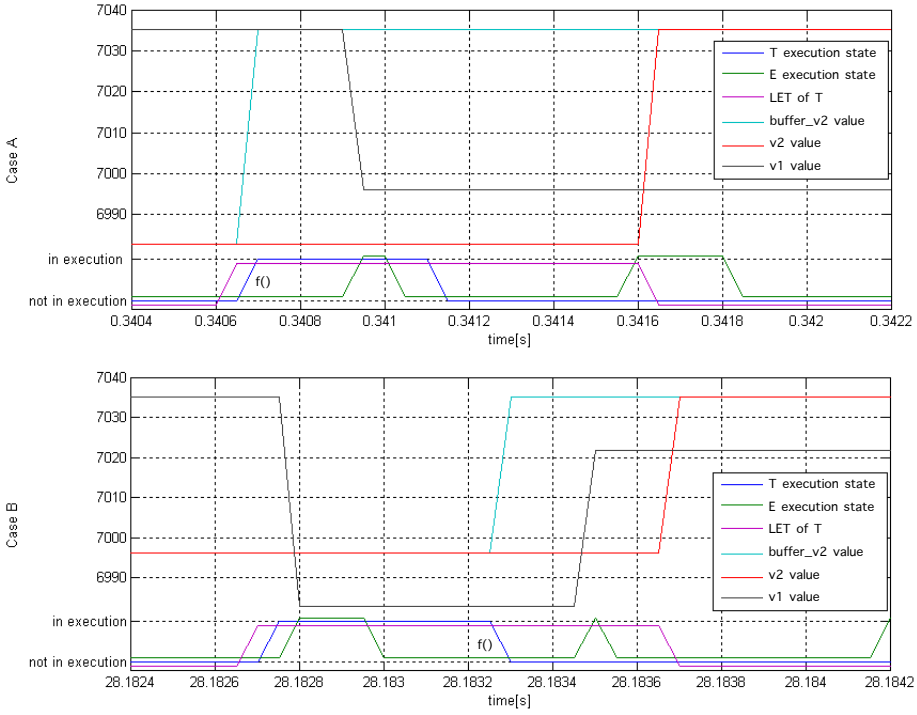

Fig. 11. The output value of v_2 is unaffected in the TDL-based system

5 Conclusions

Modeling the timing behavior of legacy applications with TDL represents an instance of bridging the gap between the general benefits advocated by Model-Based-Design approaches (such as predictability, separation of concerns, portability), and the efficiency-oriented design of legacy applications. It is a meet-in-the-middle process, with the top-down direction assumed by TDL and the bottom up direction required by the legacy application.

This paper presents an approach for applying TDL timing specifications to legacy control software, focusing on achieving the required timing behavior with minimal intervention in the original application. Thus, the paper focuses on the structure of the runtime system and instrumentation, which are automatically generated from the timing specification and from information about the legacy source code and platform. This approach has been successfully applied to a complex legacy controller system in the automotive domain. Detailed description of other important aspects such as dealing with schedulability and the actual code generation algorithm are omitted due to lack of space.

The refactored legacy system is schedulable if it can be executed such that the TDL timing specifications are satisfied, i.e., every physical execution of a

synchronous TDL task takes place in the associated LET interval. Achieving schedulability is especially difficult when asynchronous tasks have higher priorities than synchronous TDL tasks.

It is worth noting that the described TDL modeling can be applied incrementally on a legacy application, starting with a single synchronous TDL task and stepwise adding more synchronous tasks. At each step, the system can be tested for schedulability, as well as for timing and functional properties. This makes the approach feasible in practice and recommends it as the core of a structured process for incremental migration of large legacy software towards more predictable systems.

References

[1] Object Management Group: Model driven architecture (2010)

[2] Sangiovanni-Vincentelli, A.: Defining platform-based design. EEDesign of EE-Times (February 2002)

[3] Lee, E.A.: Computing needs time. Commun. ACM **52**(5) (2009) 70–79

[4] Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings. Volume 2211 of Lecture Notes in Computer Science., Springer (2001) 166–184

[5] Ghosal, A., Sangiovanni-Vincentelli, A., Kirsch, C.M., Henzinger, T.A., Iercan, D.: A hierarchical coordination language for interacting real-time tasks. In: EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software, New York, NY, USA, ACM (2006) 132–141

[6] Pree, W., Templ, J.: Modeling with the timing definition language (tdl). In: Automotive Software Workshop San Diego (ASWSD 2006) on Model-Driven Development of Reliable Automotive Services, San Diego, CA. (2006)

[7] Templ J.: TDL - Timing Definition Language 1.5 Specification. Technical report, University of Salzburg, http://www.preetec.com (2008)

[8] preeTEC: The tdl tool chain. http://www.preetec.com/ (2010)

[9] OSEK: OSEK/VDX operating system specification. http://www.osek-vdx.org/ (2010)

[10] AUTOSAR Consortium: Specification of multi-core OS architecture v1.0, AUTOSAR release 4.0 (2009)

[11] Monot, A., Navet, N., Simonot, F., Bavoux, B.: Multicore scheduling in automotive ecus. In: Embedded Real-Time Software and Systems (ERTS 2010), Toulouse, France. (2010)

[12] Farcas, C.: Towards Portable Real-Time Software Components. PhD thesis, University of Salzburg (2006)

[13] Ghosal, A., Iercan, D., Kirsch, C., Henzinger, T., Sangiovanni-Vincentelli, A.: Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code. In: Proceedings of the APGES Workshop, Salzburg, Austria (2007)

[14] Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J.: Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal **8** (1993) 284–292

[15] Joseph, M., Pandya, P.: Finding response times in a real-time system. The Computer Journal **29**(5) (May 1986) 390–395

[16] Absint: aiT worst-case execution time analyzers. http://www.absint.com/ait/ (2010)
[17] Resmerita, S., Derler, P., Lee, E.A.: Modeling and simulation of legacy embedded systems. Technical Report UCB/EECS-2010-38, EECS Department, University of California, Berkeley (Apr 2010)
[18] Derler, P.: Efficient Execution and Simulation of Time-Annotated Embedded Software. PhD thesis, University of Salzburg (2010)