Migration of legacy software towards correct-by-construction timing behavior

Stefan Resmerita, Patricia Derler, Andreas Naderlinger, Wolfgang Pree

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

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) [9] and Platform-Based Design [1] 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 [2], 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) [3] 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 observed on 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.

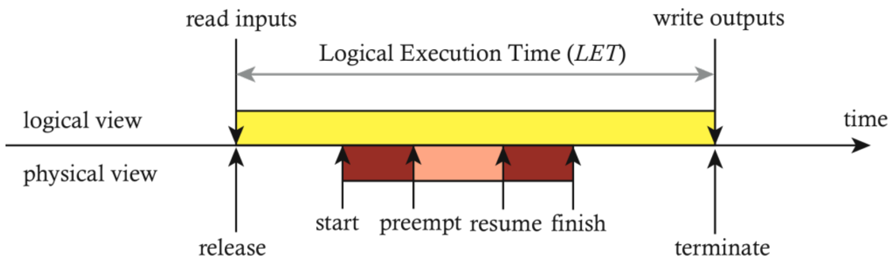


Figure 1. The Logical Execution Time (LET).

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 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 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 (platform-specific) 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 [8].

```
 module Sender {
  sensor int s1 uses getS1;
  actuator int a1 uses setA1;
  public task inc {
    input int i;
    output int o := 10;
    uses incImpl(i,o);
  }
  start mode main [period=5ms] {
    task [freq=1] inc(s1);
          //LET = 5ms (=period/freq)
    actuator [freq=1] a1 := inc.o;
    mode [freq=1] if exitMain(s1) then
                           freeze;
  }
  mode freeze [period=1000ms] {}
}
```

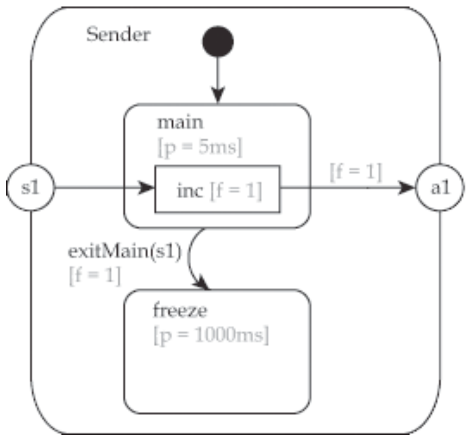


Figure 2. A TDL Example.

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 [7].

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.

We call a unit of execution of the legacy code on the execution platform a *platform task*. For example, if the legacy application runs on an OSEK [5] operating system, then a platform task is an OSEK task. 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 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 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 executions 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] 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 implementation 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.

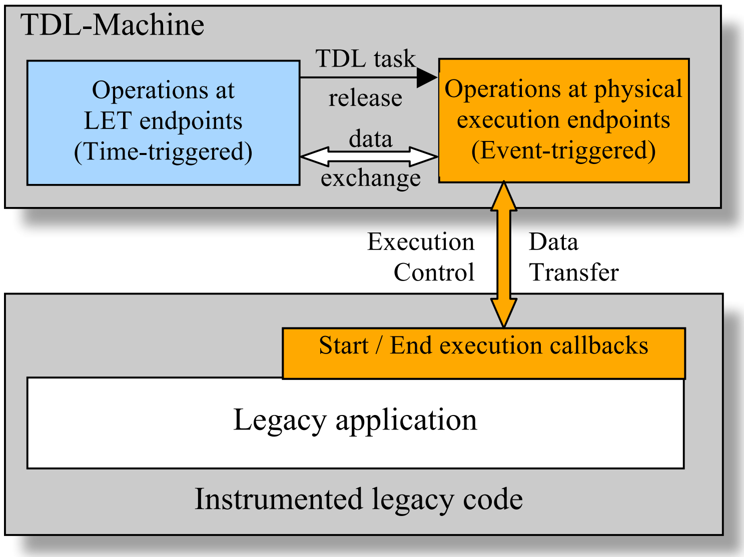
3.2 Implementation of the TDL operational semantics

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 3. 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.


TDL-Machine
Operations at LET endpoints (Time-triggered)
TDL task release
data exchange
Operations at physical execution endpoints (Event-triggered)
Execution Control
Data Transfer
Start / End execution callbacks
Legacy application
Instrumented legacy code

```
void Tx_Implementation()
{
  ... // local variables
  // execution start callback:
 On_Tx_start_execution();
  .  }
  .  }  legacy code
  .  }
  // execution end callback:
  On_Tx_end_execution();
}
```

Figure 3. The TDL-Machine architecture

3.2.1 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 the following task function:

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

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 also a LET of 2ms, but an offset of 2.4ms. An example of executions in the original system and in the TDL-modeled system is depicted in Figure 4.

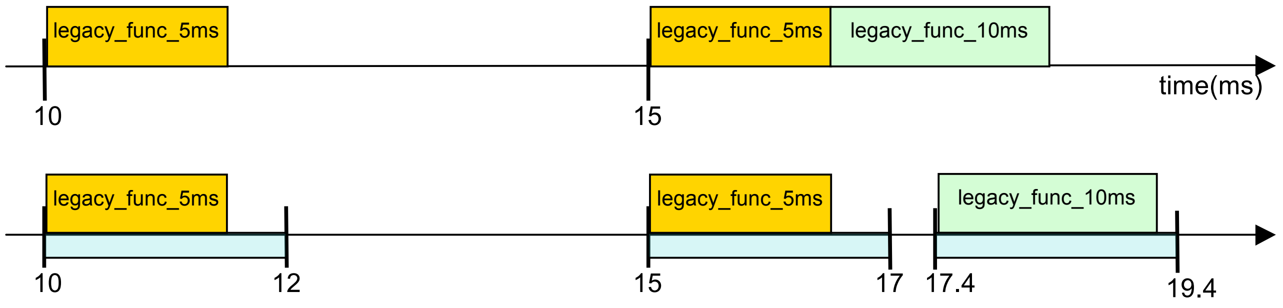


Figure 4. Execution example: original (top) and TDL-based (bottom)

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:

```
void On_T_10ms_start_execution(){      //called at the beginning of legacy_func_10ms
  WaitEvent(EV_T10_LET_START);         // blocks the execution of the platform task
}                                      //until the event is set by the TDL-Machine

void TDLMachineStep(crTime){          // time-triggered TDL-Machine  function
  ...
  if (crTime == LET_START_T10ms){
    SetEvent(Platform_Task, EV_T10_LET_START);
  }
  ...
 }
```

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

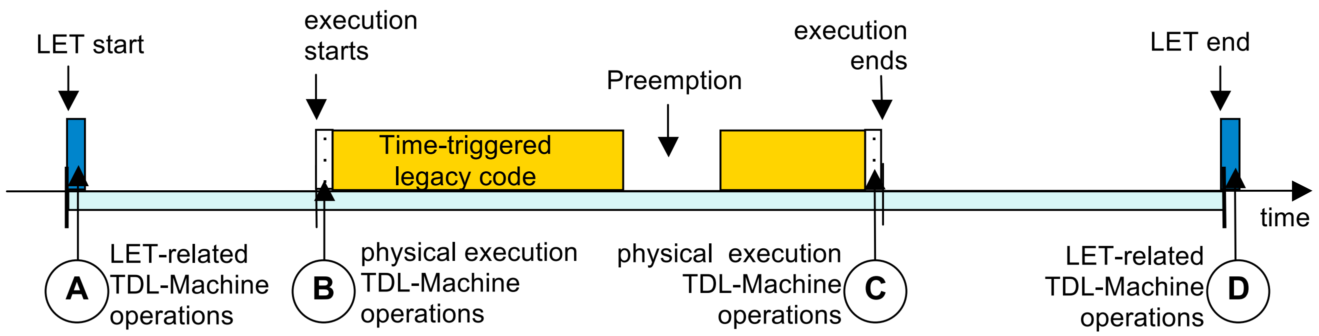
Operations A and D are executed by the time-triggered part of the TDL-Machine. To achieve a minimal instrumentation of the legacy code, we chose to implement B and C in the execution callbacks, as explained below.

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 might be defined as follows:

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

To ensure LET-based data transfer between *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*, *On_T_WRITE_end_execution,* *On_T_READ_start_execution,* and *On_T_READ_end_execution.* Figure 5 shows a sample execution trace which highlights when data transfer operations of the TDL-Machine are executed. The operations are detailed in Table 1.



LET start
execution starts
Time-triggered legacy code
Preemption
execution ends
LET end
time
A
LET-related TDL-Machine operations
B
physical execution TDL-Machine operations
physical execution TDL-Machine operations
C
LET-related TDL-Machine operations
D

Figure 5. Execution of TDL-Machine data transfer operations

Operations at LET endpoints (A and D)	Operations at physical execution endpoints (B and C)
`void TDLMachineStep(crTime){`	`void On_T_WRITE_start_execution(){`
` if (crTime == LET_START_T_READ){`	`  T_WRITE_tp_gvar = gvar;     // - B`
`   // Operation A`	`}`
`   T_READ_in_gvar = T_WRITE_o_gvar;`	`void On_T_WRITE_end_execution(){`
`   }`	`  T_WRITE_tp_o_gvar = gvar;   // - C`
	`  gvar = T_WRITE_tp_gvar;     // - C`
` if (crTime == LET_END_T_WRITE){`	`}`
`   // Operation D`	`void On_T_READ_start_execution(){`
`   T_WRITE_o_gvar = T_WRITE_tp_o_gvar;`	`  T_READ_tp_gvar = gvar;       // - B`
`   }`	`  gvar = T_READ_in_gvar;       // - B`
`}`	`}`
	`void On_T_READ_end_execution(){`
	`  gvar = T_READ_tp_gvar;       // - C`
	`}`

Table 1. Data transfer operations in the TDL-Machine for time-triggered tasks

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 Table 2.

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 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.

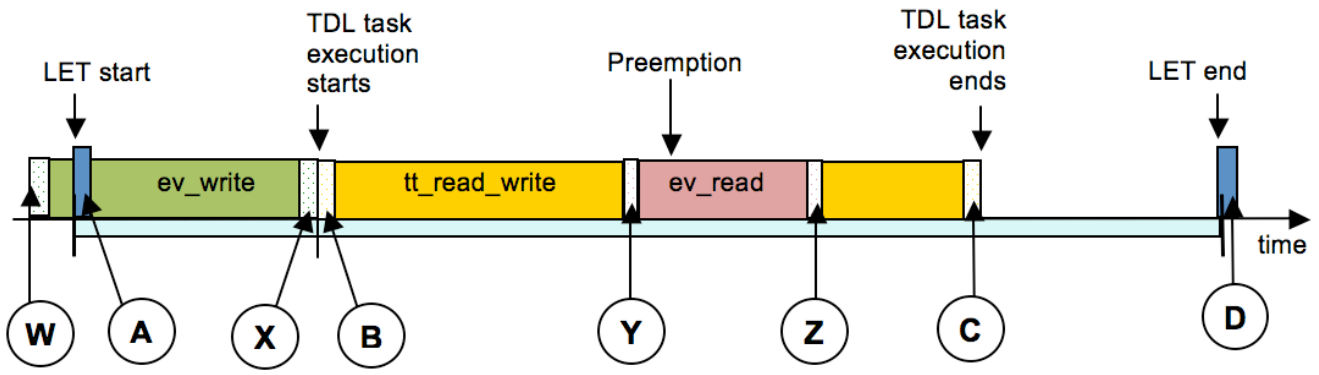


Figure 6. Mixed time and event-triggered execution example

TDL program

```
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);

  }
}
```

Operations at LET endpoints

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

Operations at physical execution endpoints

```
void On_E_WRITE_start_execution(){
  E_WRITE_tp_gvar_r = gvar_r;
}      // operation W in Figure 6

void On_E_WRITE_end_execution(){
  E_WRITE_o_gvar_r = gvar_r;
  gvar_r = E_WRITE_tp_gvar_r;
}      // operation X in Figure 6

void On_E_READ_start_execution(){
  E_READ_tp_gvar_r = gvar_r;
  E_READ_tp_gvar_w = gvar_w;
  gvar_r = E_WRITE_o_gvar_r;
  gvar_w = T_RW_o_gvar_w;
}      // operation Y in Figure 6

void On_E_READ_end_execution(){
  gvar_w = E_READ_tp_gvar_w;
  gvar_r = E_READ_tp_gvar_r;
}      // operation Z in Figure 6

void On_T_RW_start_execution(){
  T_RW_tp_gvar_r = gvar_r;
  T_RW_tp_gvar_w = gvar_w;
  gvar_r = E_WRITE_o_gvar_r;
}      // operation B in Figure 6

void On_T_RW_end_execution(){
  T_RW_tp_o_gvar_w = gvar_w;
  gvar_r = T_RW_tp_gvar_r;
  gvar_w = T_RW_tp_gvar_w;
} // operation C in Figure 6
```

Table 2. TDL model of the legacy example

4. 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.

Bibliography

[1] Alberto Sangiovanni-Vincentelli. Defining platform-based design. EEDesign of EETimes, February 2002.

[2] Edward A. Lee. "Computing Needs Time". Communications of the ACM, 52(5):70-79, May 2009.

[3] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A hierarchical coordination language for interacting real-time tasks. In EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pages 132–141, New York, NY, USA, 2006. ACM.

[4] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 166-184.

[5] OSEK. OSEK/VDX operating system specification 2.2.1. http://www.osek-vdx.org.

[6] Wolfgang Pree and Josef Templ. Modeling with the Timing Definition Language (TDL). Model-Driven Development of Reliable Automotive Services: Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Revised Selected Papers, pages 133–144, 2008.

[7] preeTEC. The TDL tool chain. http://www.preetec.com/.

[8] Josef Templ. TDL—Timing Definition Language 1.5 Specification. Technical report, http://www.preeTEC.com, 2008.

[9] Object Management Group. Model driven architecture. Technical report, http://www.gigascale.org/pubs/141.html