# Simulating Real-Time Software Components based on Logical Execution Time

**Andreas Naderlinger, Josef Templ, Wolfgang Pree**
**C. Doppler Laboratory Embedded Software Systems**
**University of Salzburg, Austria**

***firstname.lastname@cs.uni-salzburg.at***

## Abstract

Real-time software components based on the logical execution time (LET) paradigm exhibit equivalent observable behavior independent from the execution platform respectively the simulation environment. Thus, LET ensures a perfect match between simulation and execution on a potentially distributed hardware without having to consider platform specific details already in the application model. Especially for complex multi-mode multi-rate systems, a virtual machine (VM) is the favored approach to ensure the correct timing behavior. Simulation environments typically provide a trigger mechanism that allows for implementing such a VM. This paper discusses data dependency problems that may arise when simulating LET-based components and which considerably limit the applicability of existing approaches in practice. The identified shortcomings concern components with cyclic data flow, control loops involving plants without delay, and the combination of LET-based and conventional components. We present an execution mechanism based on a 2-step 3-phase VM architecture that overcomes these limitations. The presented approach is implemented in MATLAB/Simulink and applicable for mixed time- and event-triggered systems.

**Keywords:** logical execution time, data dependency, E-Machine, Simulink, TDL

## 1. INTRODUCTION

Modeling and simulation environments enable the developer to design an application step-by-step and to continuously test, analyze and optimize its behavior already early in the development process. Automatic code generators transform models typically into C code that is compiled later and executed on some target platform. But although the generated code may perfectly match the modeled functionality, when executed on an actual hardware platform, the application is likely to show slightly different or even totally unexpected behavior. On a hardware platform, the execution of some control task functionality or network communication may take a considerable amount of time, whereas the execution in a simulation environment completes basically without any time consumption. Especially for distributed systems it is common practice to relax this mismatch by introducing additional, arbitrary platform-dependent delays in the intrinsically platform neutral model. In other words, model simulation and execution of the corresponding code are only approximately the same at best and as the generated code is typically fine-tuned manually, the correspondence to the model is lost. The logical execution time (LET) [9] abstracts from a particular platform and communication topology. This allows someone to change the underlying platform and even to distribute components between different nodes without affecting the overall system behavior – provided that enough computing and communication resources are available. This enhances the status of simulation environments considerably as simulation results match the behavior on a target platform perfectly. Previous work [15] showed that simulating LET-based applications is feasible in principle using Simulink. In this paper we focus on three aspects which immediately arise in practice and which required us to refine the approach described in [15]: (1) the simulation of multiple LET-based components with cyclic data dependencies, (2) the simulation of control loops involving plants without delay, and (3) the stepwise migration of simulation models towards LET. We present an execution mechanism for the simulation environment MATLAB/Simulink [19].

## 2. TDL – A LET BASED LANGUAGE

The Timing Definition Language (TDL) [17] is a high-level software description language that allows the explicit timing specification of hard real-time components in a platform independent manner. TDL is based on the logical execution time abstraction (LET) introduced in the realm of Giotto [9].

LET means that the observable temporal behavior of a task is independent from its physical execution. It is only assumed that the physical task execution is fast enough to fit somewhere within the logical start and end points. For a particular platform this has to be ensured via an adequate scheduling given the worst case execution time for each task. The inputs of a task are read at the *release* time and the newly calculated outputs are available at the *terminate* time. Between these two logical instants, the outputs have the value of the previous execution. Although LET introduces additional response time overhead, it provides the cornerstone to deterministic behavior, platform abstraction, and well-defined interaction semantics between parallel activities [7].

In order to support complex applications, TDL introduces a component model. Applications can be decomposed into individual components, each represented as a TDL module.

```
module S {

 import R;

 sensor
  int s uses getS;

 actuator
  int a uses setA;

 public task send {
  input int i;
  output int o;
  uses sendImpl(i,o);
 }
 start mode main [period=10ms] {
  task
   [freq=1] send(s);
  actuator
   [freq=1] a := send.o;
  mode
   [freq=1] if exitMain(R.rcv.o)
              then freeze;
 }

 mode freeze [period=1000ms] {
 }
}
```

```
module R {

 import S;

 public task rcv {
  input int i;
  output int o;
  uses rcvImpl(i,o);
 }
 start mode main [period=5ms] {
  task
   [freq=1] rcv(S.send.o);
 }
}
```

**Figure 1.** TDL module S and R

```
[000] call 1 //actuator init: setA(a)
[001] return //end of initialization
      //mode freeze
[002] future 4, 1000000
[003] return
[004] nop 1//eot, end of task terminations
[005] nop 2//eoa, end of actuator updates
[006] jump 2 //next cycle: freeze
      //mode main
[007] call 2 //get: s := getS()
[008] call 3 //release task: send
[009] release 0 //uses: sendImpl
[010] future 12, 10000
[011] return
[012] call 0 //terminate task: send
[013] nop 1 //eot, end of task terminations
[014] call 4 //actuator update: a := o
[015] call 1 //actuator setter: setA(a)
[016] nop 2 //eoa, end of actuator updates
[017] if 0, 20 //mode switch guard: exitMain
[018] call 5 //mode switch driver
[019] switch 0 //mode switch -> freeze:0
[020] jump 7 //next cycle: main
```

**Figure 2.** E-Code for the TDL module S

**Module.** A module performs computations represented as tasks and communicates with the environment by means of sensors and actuators. Modules may import one or multiple other modules and access their public entities. At runtime, all modules of an application run in parallel. Thus, a TDL application is the parallel composition of a set of TDL modules, which are synchronized to a common time base. A module is in exactly one mode at a time.
**Mode.** A mode is a particular operational state of a module. Modes have a period and consist of a set of activities. An activity can be a task invocation, an actuator update, or a mode switch. Each activity specifies its own logical execution instants relative to the mode period. The LET of a task is always greater than zero, whereas actuator updates and mode switches are executed in logical zero time.
**Task.** A task represents a computational unit of an application. It declares an external function, which can be implemented in an imperative language such as C. Furthermore, a task declares input, state, and output ports.
**Port.** Ports are typed variables used for data communication. TDL differentiates between task ports, sensors and actuators.
**Guard.** A guard is an external boolean function. Guards may be used to execute mode activities conditionally.

Figure 1 shows a sample TDL application consisting of two modules $S$ and $R$. Module $S$ (sender) has two operational modes $main$ and $freeze$. While $freeze$ does not perform any activity, $main$ invokes the task $send$ once per mode period (LET=10ms) and updates the actuator $a$. The task $send$ reads a sensor value and provides its output to the actuator. Once per mode period, $S$ evaluates the mode switch guard $exitMain$ which uses the imported port $rcv.o$ from module $R$. Module $R$ (receiver) has a single mode $main$ that executes task $rcv$ once every $5ms$ (LET=5ms). $R$ imports module $S$ to use the port $S.send.o$ as input for $rcv$.

## 2.1. Executing TDL Components

The concept of the E-Machine (Embedded machine) was first introduced in the realm of Giotto [10]. The E-Machine is a virtual machine (VM) that lays the foundation for platform-independent real-time software, while the implementation of the VM itself depends on the platform (e.g. the operating system). The E-Machine ensures the correct timing behavior of real-time applications. Therefore, the timing behavior described in TDL components is compiled into an intermediate code, called the Embedded Code [10] or E-Code for short. E-Code describes an application's reactivity, i.e. time instants to release or terminate tasks or to interact with the environment. It is a portable description as it relies on logical timing and is independent of a particular platform. E-Code is represented as a sequence of instructions that are interpreted by the E-Machine [6, 10]. Figure 2 lists the E-Code for the TDL sample module $S$. According to the E-Code instructions the E-Machine timely hands tasks to a dispatcher and executes drivers. A driver performs communication activities, such as reading sensor values, providing input values for tasks at their release time or copy output values at their termination time. TDL compiler plug-ins generate drivers automatically for a certain platform according to the language binding rules [17]. They are part of the so-called glue code which additionally comprises type declarations, etc., as well as the communication layer [5]. The functionality code, i.e. task, guard, initializer, sensor and actuator functions, has to be provided separately.

## 2.2. TDL E-Code Instruction Set

The E-Code instruction set for TDL builds on the set defined for the Giotto language. A TDL E-Code instruction $c(args)$ consists of the command $c$ and its list of integer arguments $args$. All commands are executed synchronously: $call(d)$ executes the driver $d$; $release(\tau)$ marks the task $\tau$ as ready for execution; $future(a, \delta)$ plans the execution of the E-Code block starting at address $a$ in $\delta$ $\mu$s; $if(g, a_{else})$ proceeds with the next instruction if $g$ evaluates to $true$ else jumps to $a_{else}$; $jump(a)$ jumps to the E-Code at address $a$; $switch(m)$ performs a mode switch to mode $m$, i.e. the E-Machine continues with the first instruction of $m$; $return$ terminates an E-Code block; $nop(f)$ is a dummy (no operation) instruction, where $f$ is used as a marker for identifying different sections in the E-Code (see section 4.1.).

## 2.3. E-Code Blocks

An E-Code block is a list of E-Code instructions terminated by a $return$ instruction. It specifies for one logical time instant within a mode period the actions that must be taken by the E-Machine in order to comply with the LET semantics. For a logical time instant $t$, the following sequence of actions comprises one E-Code block: (1) update output ports of task invocations logically terminating at $t$ with the result values from their execution; (2) update actuators that are defined to be updated at $t$; (3) switch mode if a mode switch is defined at $t$; (4) update input ports of tasks that are defined to be released at $t$; (5) release tasks that are defined to be released at $t$; (6) advance time to the next logical time instant $t + \delta$; (7) return from E-Code block.

Note that sensors are read whenever their value is required. However, at one particular logical time a sensor is read only once, even if it is used multiple times.

## 2.4. Bisection of E-Code Blocks

As a generalization of the original use of E-Code in Giotto, TDL supports applications consisting of multiple components (modules). All modules of an application are (logically) executed in parallel and share the same logical time. For a single-threaded execution of multiple independent modules it suffices to execute an *E-Machine step* for all modules sequentially in any order. An E-Machine step for a module means to execute the current mode's E-Code block that belongs to the current logical time. In addition to independent modules, TDL allows one to define a data flow between modules by means of reading from an imported output port. The execution order then influences the application behavior because port updates in a certain module are only visible to modules executing afterwards. For import relations conforming to a directed acyclic graph (DAG), it suffices to sort the set of modules topologically according to the import relationship [5]. For modules with cyclic import dependencies, which are also allowed in TDL, such a sort is not possible. Therefore, the single-threaded execution of modules must be split into multiple phases instead of sorting. In the first phase, all output ports must be updated before any other module reads them in the second phase. Each phase is represented by an individual section within an E-Code block. The TDL compiler inserts a $nop(f)$ instruction with an appropriate flag $f$ to mark the end of a section.

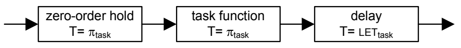

**Figure 3.** LET semantics in a data flow diagram

## 3. SIMULATING TDL COMPONENTS

A major advantage of the LET abstraction is the identical behavior of real-time applications on any platform that is powerful enough. This makes simulation of LET-based applications particularly useful. Most simulation environments fail in considering non-functional properties such as timing, in particular for distributed applications. Typically, simulation results can only be seen as estimations for the actual behavior on a real hardware target. By considering a simulation environment as yet another platform, the LET assumption guarantees an equivalent behavior. Thus, the behavior of a model simulation exactly matches the behavior shown in practice. As communication latencies are also subsumed under the LET, the simulation is totally independent from the intended communication topology.

To achieve LET semantics of a task in a simulation environment, the simulation engine must ensure that (1) the task is released and terminated at well-defined logical points in time, that (2) inputs are only read once when the task is released, and (3) output ports are solely updated when the task terminates logically. Figure 3 shows a data flow diagram that exhibits such semantics. The task function block which implements the task's functionality has a sample time $T$ equal to the period $\pi$ of the task. Additionally, it is surrounded by a zero-order hold and a discrete delay block. The sample time of the zero-order hold block is set to the task's period, which ensures that input values do not change while the task executes. The sample time of the delay block is set to the LET of the task, which ensures that the newly calculated output values are available only after the task terminated logically.

This approach is straightforward, however, it fails when mode switching logic and multiple execution rates come into play. Typically, control systems consist of periodically executed tasks and involve mode-switching logic [9]. Depending on the current mode, the application executes individual tasks with different timing constraints or even changes the set of executed tasks. It is difficult to understand the overall behavior of the model as it becomes too cluttered for all but trivial applications. It is also unclear if the exact mode switch semantics can be obtained at all [16]. Our work is based on the approach described in [15] which uses the concept of a virtual machine (E-Machine) [10] that coordinates the timing.

## 3.1. Simulink Background

Basically, executing a block diagram involves repeatedly (1) solving all block output equations, (2) updating the states, and (3) advancing time. In principle, there exist different strategies for (1). The blocks could be executed in arbitrary order until the outputs do not change anymore, i.e. a fixed point is reached. Although this approach requires only very little information about block internals [12], it is rather inefficient. Simulink therefore follows a different strategy. A sorted block order is derived from the initialization of the simulation, which avoids the need for iteration. At each simulation step, the blocks are executed in this fixed order that does not necessarily comply with the block connections, but also depends on the feedthrough characteristics of each block. A block with *direct feedthrough* must know its current input values before it can set its output values. Assuming a model with discrete blocks only, at each simulation step, the Simulink engine computes the model's outputs by invoking the output method of each block (*mdlOutputs*) in the sorted order derived during the initialization. Afterwards, the engine computes the state of the model by invoking the update method of each block (*mdlUpdate*), again in the predefined order.

In the regular case, the execution order of blocks is under control of the simulation environment. This is not true for so-called *conditionally executed* blocks respectively subsystems such as *function-call* or *triggered subsystems*. They are not part of the sorted block order. Instead, they are executed in the context of their triggering or function-call initiating block and thus kind of inherit the position in the sorted block order. This position of the trigger block, however, is determined by feedthrough characteristics and data dependencies of itself and of all the blocks it triggers. A *function-call initiator* autonomously decides which blocks to execute and in which order.

The built-in Simulink block set can be extended by so-called S-Functions, which are Simulink blocks implemented in a programming language such as C. S-Function blocks are allowed to trigger the execution of function-call subsystems by using the Simulink macro *ssCallSystemWithTid* in their *mdlOutputs* method. After the function call subsystem was executed, the control is returned to the S-Function which resumes execution.

## 3.2. The E-Machine as an S-Function

In order to use a typical state-of-the-art simulation environment as a platform for TDL, the chosen integration approach must match with the computational model of the simulation tool. In the Ptolemy [4] environment, for example, one can

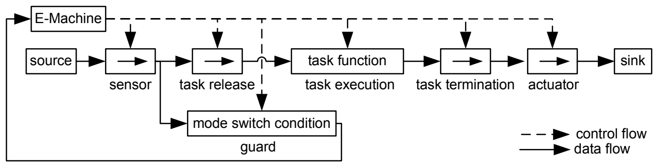

**Figure 4.** The basic principle of an E-Machine as a Simulink S-Function and triggered subsystems

implement customized directors that ensure the desired behavior. Commercial tools such as Simulink are typically more restrictive. Simulink's model of computation (MoC) is based on continuous time. It is rather complex and there exists no formal definition; the implementation is hidden in the simulation engine [1, 2]. A straight forward modeling of TDL components with standard Simulink blocks is not feasible, especially if they comprise several modes [16]. In principle, there are several approaches for achieving LET semantics in Simulink [15]. We build on the most promising approach, which is based on the E-Machine concept.

The E-Machine described in section 2.1. is realized in Simulink by means of an S-Function that contains the E-Code interpreter. Task-, guard-, and initializer functions are implemented using function-call subsystems, which have to be modeled by the application developer using ordinary Simulink blocks. The glue code, i.e. drivers that perform communication activities, such as reading sensors or copying values from task ports to actuators, are also realized as function-call subsystems. They all get invoked by the E-Machine. The output ports of such a driver subsystem are directly connected to its input ports, which corresponds with assignments in the imperative programming paradigm as soon as the system is triggered. The TDL tool-chain generates the driver subsystems automatically when the simulation is started. Only the timing, i.e. the TDL description, has to be specified and the functionality, e.g. task function subsystems, has to be modeled. Section 5. gives an overview of the development process with TDL in Simulink.

Figure 4 exemplifies this E-Machine approach for a simplified application. The placement of the individual blocks conforms to the data flow, which is basically from left to right along the arrows from a source to a sink. The source value is read by a sensor which provides the value to a guard and a task. The actuator block uses the output port of a task to write to a sink. Regarding execution order, the figure must not be read from left to right. The E-Machine triggers the individual blocks according to the E-Code resulting in the above order of activity execution (see section 2.3.). This also ensures the correct LET behavior of a task by triggering its release and termination drivers at the right time instants.

When the E-Machine triggers the execution of a guard, it immediately reads the result via an input port. The result influences the further execution of E-Code instructions and consequently which subsystem to trigger. Thus, the E-Machine block has direct feedthrough.

The E-Machine has to be invoked whenever the simulation time matches the logical time of a TDL activity as defined in the E-Code. To ensure this, we use a fixed-sample time for the E-Machine (*E-Machine period*), which is the GCD of all activity periods.

## 3.3. Data Dependency Problems

The S-Function implementation of the E-Machine for a simulation environment is very close to the original E-Machine concept of Giotto. Compared to other approaches, this results in a straightforward and efficient simulation model [15]. However, due to data dependency problems which can occur in simulation environments, its practical applicability turned out to be limited. We identified the following application scenarios which in general cannot be handled by this previous approach: (1) Cyclic import relationships between LET-based components (controllers), (2) control loops involving plants without delay, and (3) mixed LET-based and conventionally modeled controllers. Theses cases are discussed in more detail below. They are all related to cyclic data flow and the ability of the simulation environment to find a valid strategy for executing each individual block. While some models of computation support cycles without a delay (e.g. the Ptolemy synchronous/reactive domain), Simulink[1] (as well as LabView or the Synchronous Data Flow domain in Ptolemy) doesn't. Delays are introduced by explicit delay blocks, or by other blocks whose inputs have indirect feedthrough. Indirect (or non-direct) feedthrough means that the block's output $y$ depends on the state $x$, but is not directly controlled by the input $u$. The delay block for example is described by the following state space description $y(t) = x(t)$, $x(t + \Delta) = u(t)$ where $\Delta$ is the delay (i.e. the sample time of the block).

**Cyclic import relationships.** Import relationships lead to Simulink signals between driver blocks of different modules. For example, the termination driver of task $send$ in module $S$ is connected to the release driver of task $rcv$ in module $R$. As outlined in section 3.1., Simulink uses a fixed block update order in which blocks are executed. This rules out the approach described in [15] using one E-Machine per module, as relative to other blocks, all drivers of a particular module are executed at once. Only one global E-Machine, which interprets the E-Code of all modules, leads to a model that follows LET semantics and is resolvable by the simulation engine.

**Plants without delay.** Closed-loops are well known concepts in control theory (e.g. for PID controllers). The controller monitors the output of the plant, i.e. the system under control, via sensors and adjusts its actuators to achieve a specified response. Plants with direct feedthrough, i.e. without introducing a delay, must – as a whole – execute after actuators are updated and before sensors are being read. With the described E-Machine approach both sensors and actuators are under control of one single S-Function block. Consequently, Simulink cannot find a valid update order of blocks.

[1]Simulink's support for cycles without delays (algebraic loops) is quite limited [3].

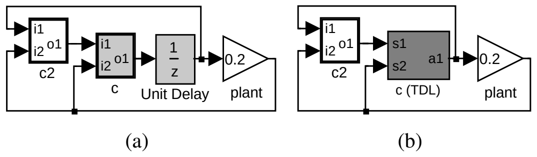

**Figure 5.** A Simulink model (a) before and (b) after migrating the atomic subsystem of the controller $c$ to TDL

**LET- and Non-LET-based controllers.** Another scenario concerns the interaction between TDL modules (controllers) and controllers that are not LET-based, for example when a control application is migrated towards LET step-by-step. Typically, controllers are modeled as *atomic* (nonvirtual) subsystems in Simulink in order to reflect the desired software partitioning in the simulation and the program synthesis. The equations defined by an atomic subsystem are evaluated as a unit. Usually, in order to avoid 'false' algebraic loops [13] and to approximately mimic the computation time on a real hardware platform, unit delay blocks are added between controllers and the plant and also between multiple controllers. Figure 5(a) shows a sample model with two conventionally modeled controllers and a plant. Figure 5(b) shows the same model partially based on LET. The controller $c$ is replaced by a TDL module executing the former implementation as a task. The unit delay is now implicitly replaced by the task's LET ensured by the E-Machine. Again, Simulink cannot find a valid update order of blocks.

In any of these cases, Simulink reports a *data dependency violation*[2], which is similar to an *algebraic loop error*. From the control engineer's point of view, this appears to be counterintuitive, since the LET of a task is always greater than zero and thus should introduce the required delay.

## 4. 2-STEP E-MACHINE ARCHITECTURE

This section describes a mechanism for simulating LET-based software components, which is also based on an E-Machine, but does not suffer from data dependency problems as described above. We shall start with new requirements concerning the E-Code representation and continue with its execution strategy during the simulation. The described mechanism was implemented in Simulink. However, we expect it to be applicable for a larger number of simulation environments. Finally, we shall shortly sketch a minor extension of this approach to deal with both, time- and event-triggered systems.

[2]In some rare cases, Simulink may be able to simulate the model, when the *block reduction optimization* option is enabled.

## 4.1. Trisection of E-Code Blocks

Data dependencies among TDL modules (see section 2.4.) require the partitioning of E-Code into two sections. However, in order to execute the plant or other non-TDL blocks between setting actuators and reading sensors, E-Code must be split into three disjoint sections. These E-Code sections represent the following three phases:

**tt - Task termination phase.** The *tt* phase includes all activities that must be carried out in order to make output ports available at the LET end of a task invocation. After that, the updated output ports are visible to client modules and may be used as input for any other activity.

**au - Actuator update phase.** The second phase *au* includes all activities that must be carried out in order to update actuators with new values, potentially including the evaluation of guards.

**mstr - Mode switch and task release phase.** The last phase *mstr* includes all activities that must be carried out in order to perform mode switches and to release new task invocations. Releasing a task invocation means to copy the actual parameters (stemming from sensors or task ports) to the input ports of the released task. This phase also comprises the execution of task functions.

Each phase must be executed for all TDL modules. This E-Code trisection is the basic requirement for simulating models with data dependencies as described above. From these dependencies, we can immediately derive the execution order of our three phases and the remaining blocks: (1) *tt*, (2) *au*, (3) non-TDL blocks, (4) *mstr*. To identify the individual phases at runtime we introduce *markers (tags)* in the E-Code that separate the corresponding sections from each other. Markers are represented as *nop* instructions with an appropriate flag: *eot* indicates the end of the *tt* section (phase), *eoa* indicates the end of the *au* section (phase).

It should be noted that a trivial solution to the data dependency violation is to put also all non-TDL blocks under the control of the E-Machine. This would basically result in a $4^{th}$ E-Machine phase which is executed between *au* and *mstr* by an E-Machine trigger. However, this contradicts the understanding of an independent plant and brings about several drawbacks; above all, the limitation to no longer support continuous Simulink blocks. Hence, we did not pursue this option.

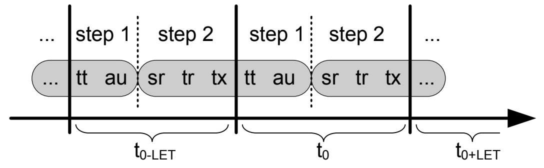

**Figure 6.** Separation of triggers into steps

## 4.2. The 2-Step Execution Mechanism

Our primary goal is to decouple the triggering of actuators and sensors in order to execute non-TDL blocks in between. Figure 6 shows the execution of the individual triggers along the timeline for a simple example where a task reads from a sensor and updates an actuator: at time $t_{0-LET}$, the sensor is read (*sr*), the task is released (*tr*) and executed (*tx*); at time $t_0$, the task is terminated (*tt*) and the actuator is updated (*au*). Afterwards, but at the same logical time, the next invocation begins with executing *sr*, *tr*, and *tx*. The basic idea of this approach is to split the execution of all triggers for a particular logical time into two separate steps. Step 1 executes *sensor-independent activities* such as the termination of tasks, while step 2 executes activities that are potentially *sensor-dependent* such as the release driver of a task. Each step is executed from a different S-Function, so that the plant or other non-TDL blocks can be executed in between:

**E-Machine 1** executes step 1, which comprises the execution of (1) task termination drivers, (2) actuator drivers, and (3) their guards if both do not depend on a sensor value; additionally, the E-Machine 1 executes (4) port initializer functions when the simulation starts.

**E-Machine 2** executes step 2, which comprises the execution of the following activities: (1) sensor drivers, (2) mode switch drivers, (3) mode switch guards, (4) task release drivers, (5) task execution drivers, and (6) task guards; additionally, (7) actuator drivers and (8) their guards, if the actuator itself or the guard depends on a sensor value[3].

According to this scheme, the glue-code generator distributes the list of drivers (function-call subsystems) among the two E-Machines. Both E-Machines operate on the same E-Code and both are based on the same implementation. They only differ in their *mdlOutputs* function. Algorithm 1 shows the implementation for E-Machine 1, which executes only phase *tt* and phase *au*.

**Algorithm 1** *mdlOutput* function of E-Machine 1

```
for (module $m$ : $modules$) { // step 1, phase $tt$
 $f_t[m] \leftarrow$ readInput() //read future time from E–Machine 2
 if ($f_t[m] = now_\epsilon$){ //logical time = simulation time?
  $f_a[m] \leftarrow$ readInput() //read future addr. from E–Machine 2
  $a \leftarrow f_a[m]$
  //execute E–Code instr. for $m$ at $a$ until $nop_{eot}$
  interpreteECode($m$, $a$, $nop_{eot}$)
 }
}
for (module $m$ : $modules$) { // step 1, phase $au$
 if ($f_t[m] = now_\epsilon$){ //logical time = simulation time?
  //execute E–Code instr. for $m$ at $a$ until $nop_{eoa}$
  interpreteECode($m$, $a$, $nop_{eoa}$)
 }
}
```

[3]If an actuator or its guard reads a sensor, Simulink is only able to simulate the model, if there is a delay introduced somewhere along the signal path between actuator and sensor.

**Algorithm 2** *mdlOutput* function of E-Machine 2

```
for (module $m$ : $modules$) { // step 2
  if ($f_t[m] = now_\epsilon$){ //logical time = simulation time?
    //execute E–Code instr. for $m$ at $f_a[m]$ until $return$
    interpreteECode($m$, $f_a[m]$, $\perp$) //sets new $f_a[m]$, $f_t[m]$
    writeOutput($f_t[m]$) //write future time to E–Machine 1
    writeOutput($f_a[m]$) //write future address to E–Machine 1
  }
}
```

Algorithm 2 shows the implementation for E-Machine 2, which passes through the whole E-Code in order to execute phase *mstr* and to properly handle sensor-dependent actuators which appear in phase *au*.

Splitting the E-Code interpretation of one module into separate E-Machines introduces additional synchronization requirements. Mode switches performed by E-Machine 2 have to be signaled to E-Machine 1. More precisely, E-Machine 1 reads the $a$ and $\delta$ argument of the last $future$ instruction of E-Machine 2 via a Simulink signal to timely resume execution at the correct E-Code instruction.

After executing step 2, time passes because the LET of a task is always greater than zero. However, Simulink is not aware of this, as it is not apparent from the data flow. Consequently, without any further arrangements, Simulink could not derive a valid block update order and would still report a *data dependency violation*. Whenever data flow is implicitly delayed by the E-Machine, i.e. a future instruction is hit, the delay must be reflected in the simulation model. Time passes between the execution of a task and its termination, as well as between the E-Machine executions of step 2 and step 1. Thus placing a *unit delay block* between each *tx* and *tt* driver pair and between the two E-Machine blocks enables Simulink to find a valid block update order, that exactly follows the LET semantics. The sample time of the delay blocks is set to the E-Machine period. Effectively, they have no impact on the observable timing behavior.

Figure 7 illustrates the 2-step E-Machine architecture and the block update order of the overall system. The introduced delay blocks between the task execution and the termination driver and between E-Machine 2 and E-Machine 1 are executed first. Afterwards, the simulation environment executes E-Machine 1. The plant (or any other non-TDL block) executes third. Hereafter, the E-Machine 2 executes step 2.

To avoid illegal data dependencies caused by cyclic import relationships, all TDL modules in a simulation model have to be controlled by one single E-Machine pair only.

## 4.3. Mixed Time/Event-Triggered Systems

So far we only considered purely time-triggered real-time applications. Recently, TDL was extended by a notion for asynchronous (event-triggered) activities with a well-defined synchronization mechanism for data flow between

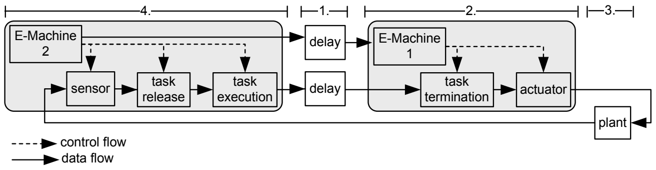

**Figure 7.** The 2-step E-Machine architecture

time- and event-triggered activities [18]. The presented 2-step E-Machine architecture proved to fit well for the new requirements. To ensure that the different priorities are preserved and that the data flow exactly follows TDL semantics, also asynchronous activities are triggered by the E-Machine, which now uses an inherited sample time. However, when using asynchronously executed TDL activities, simulation looses its property to exactly match with the behavior on an arbitrary potentially distributed hardware platform. Platform specifics are not modeled in the simulation, which is thus neither aware of any target specific scheduling mechanism, network topology, nor expected execution or transmission times, etc. Nevertheless, the simulation is still useful for analyzing synchronous and asynchronous TDL activities together with the plant model.

## 4.4. Evaluation

We evaluated the runtime overhead of the presented E-Machine architecture compared to a single-step approach. For a model executing 200 empty task functions at an average rate of 30ms ($\approx$ 4400 E-Code instructions), we measured about 8% performance loss. In an active rear steering case study, the overhead was about 6%. In general, the overhead decreases with rising complexity of the plant and the task functions. Furthermore, it increases with the length of the E-Code sections *tt* and *au*, because the E-Machine 2 iterates over the whole E-Code for executing sensor-dependent actuators. If required, this can be considerably optimized at the expense of analyzing the E-Code during initialization.

## 5. CODE GENERATION & TDL TOOLS

The 2-step E-Machine approach has been implemented in the TDL tool-chain [14]. The timing behavior of the application, i.e. the TDL description, is specified by means of the TDL:VisualCreator, a graphical modeling tool that is integrated into Simulink via a *TDL Module Block* in the Simulink library. The functionality and the plant are modeled with Simulink blocks. For simulating the model, the TDL description is compiled into E-Code and automatically transformed into interconnected Simulink blocks which also link to the already modeled functionality blocks. During the simulation, the E-Machine pair triggers the individual blocks according to the E-Code. Once the simulation exhibits satisfactory behavior, we can go about generating code. Therefore, we use the TDL:VisualDistributor tool to define a hardware topology and to map the TDL modules to their target nodes. This also requires to specify worst-case execution times and hardware devices for sensors respectively actuators. A flexible plugin-based code generation framework generates the required C glue code and, in case of a distributed system, the required communication schedule according to the specified target platform [14]. We use MathWork's Real-Time Workshop Embedded Coder [19] to generate C code for the control task functionality. The generated code can now be compiled and linked with the platform specific E-Machine.

## 6. RELATED WORK

The concept of LET was introduced in the realm of Giotto [9]. Throughout this paper we used one of its successors, the Timing Definition Language (TDL) [17], as a language based on the LET programming model.

The foundation of our work, i.e. the initial prototypical E-Machine S-Function implementation, is described in [15]. It uses one single-step E-Machine for each module and is thus prone to data dependency violations.

TrueTime [8] provides a MATLAB/Simulink toolbox for simulating distributed real-time control systems. The resulting Simulink model, however, is tailored to one specific hardware platform and network topology.

Mosterman et al. [13] describe illegal loops in Simulink that arise when partitioning models in order to match the hardware platform. The authors propose an interleaved execution mechanism to resolve a certain class of algebraic loops involving atomic subsystems. The simulation of LET-based components requires solving data dependencies involving triggered subsystems.

Also another Giotto extension, HTL, allows for the simulation of LET-based systems in Simulink [11]. In contrast to our approach, the simulation results do not match exactly the LET description. For breaking algebraic loops, additional delay blocks are introduced which influence the observable timing. Additionally, this implementation trades off accuracy for performance since it requires the sample rate of some blocks to be at least one decimal order of magnitude higher than actually required by the HTL description.

TDL has also been ported to Ptolemy [4], a modeling and simulation environment for heterogeneous systems, as a new model of computation. The timing behavior is described within the simulation model, whereas our approach abstracts from both, the simulation respectively the execution platform.

## 7. CONCLUSION

We presented a mechanism for simulating LET-based real-time components. We pointed out the problems with data dependencies respectively algebraic loops that typically occur when LET-based components (controllers) are simulated.The described approach solves these problems and ensures that the simulation in a modeling environment such as MATLAB/Simulink exhibits exactly the same behavior as the execution of the generated code on any potentially distributed hardware platform.

**Acknowledgements.** We would like to thank Stefan Resmerita for pointing us to the problem of simulating LET-based controllers in models involving plants without a delay, which led to the current version of the E-Machine architecture. Gerald Stieglbauer helped us with subtle points regarding Simulink's semantics.

## REFERENCES

[1] M. Baleani, A. Ferrari, L. Mangeruca, A. L. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff. Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In *DATE '05*. IEEE Computer Society, 2005.

[2] L. Carloni, M. Benedetto, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli. Modeling Techniques, Programming Languages and Design Toolsets for Hybrid Systems. Technical Report Columbus Project, IST-2001-38314 WPHS, 2004.

[3] B. Denckla. Many cyclic block diagrams do not need parallel semantics. *SIGPLAN Not.*, 41(8):16–20, 2006.

[4] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):127–144, January 2003.

[5] C. Farcas. *Towards Portable Real-Time Software Components*. PhD thesis, University of Salzburg, 2006.

[6] C. Farcas and W. Pree. Virtual execution environment for real-time TDL components. In *ETFA '07: Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation*, 2007.

[7] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. *SIGPLAN Not.*, 40(7):31–39, 2005.

[8] D. Henriksson, A. Cervin, and K.-E. Arzen. TrueTime: Real-time Control System Simulation with MATLAB/Simulink. In *Proceedings of the Nordic MATLAB Conference*, 2003.

[9] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, January 2003.

[10] T. A. Henzinger and C. M. Kirsch. The embedded machine: predictable, portable real-time code. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 315–326, New York, NY, USA, 2002. ACM.

[11] D. Iercan and E. Circiu. Modeling In Simulink Temporal Behavior of a Real-Time Control Application Specified in HTL. *Journal of Control Engineering and Applied Informatics (CEAI)*, 10(4):55–62, 2008.

[12] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In C. M. Kirsch and R. Wilhelm, editors, *EMSOFT*, pages 114–123. ACM, 2007.

[13] P. J. Mosterman and J. E. Ciolfi. Interleaved Execution to Resolve Cyclic Dependencies in Time-Based Block Diagrams. In *CDC '04: Proceedings of the 43rd IEEE Conference on Decision and Control*, 2004.

[14] A. Naderlinger, J. Pletzer, W. Pree, and J. Templ. Model-Driven Development of FlexRay-Based Systems with the Timing Definition Language (TDL). In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, Washington, DC, USA, 2007. IEEE Computer Society.

[15] G. Stieglbauer. *Model-based Development of Embedded Control Software with TDL and Simulink*. PhD thesis, University of Salzburg, 2007.

[16] G. Stieglbauer and W. Pree. Visual and Interactive Development of Hard Real Time Code, January 2004. Automotive Software Workshop San Diego (ASWSD).

[17] J. Templ. Timing Definition Language (TDL) 1.5 Specification. Technical report, University of Salzburg, 2007. Available at http://www.softwareresearch.net.

[18] J. Templ, J. Pletzer, and A. Naderlinger. Extending TDL with Asychronous Activities. Technical report, University of Salzburg, 2008. Available at http://www.softwareresearch.net.

[19] The MathWorks. http://www.mathworks.com.